

---

# Multimedia User Guide

发行版本 (HEAD detached at 9f3e22c)

SOPHGO

2024 年 07 月 25 日

# 目录

1	声明	2
2	多媒体用户手册	4
2.1	SOPHGO 多媒体框架介绍	4
2.1.1	简介	4
2.1.2	BM1686 硬件加速功能	6
2.1.3	硬件内存分类	7
2.1.4	框架之间转换	8
2.2	SOPHGO OpenCV 使用指南	11
2.2.1	OpenCV 简介	11
2.2.2	数据结构扩展说明	12
2.2.3	API 扩展说明	12
2.2.4	硬件 JPEG 解码器的 OpenCV 扩展	32
2.2.5	OpenCV 与 BMCV API 的调用原则	35
2.2.6	OpenCV 中 GB28181 国标接口介绍	35
2.3	SOPHGO FFMPEG 使用指南	39
2.3.1	前言	39
2.3.2	硬件视频解码器	39
2.3.3	硬件视频编码器	40
2.3.4	硬件 JPEG 解码器	43
2.3.5	硬件 JPEG 编码器	44
2.3.6	硬件 scale filter	44
2.3.7	AVFrame 特殊定义说明	47
2.3.8	硬件加速在 FFMPEG 命令中的应用示例	51
2.3.9	通过调用 API 方式来使用硬件加速功能	59
2.3.10	硬件编码支持 roi 编码	59
2.4	SOPHGO LIBYUV 使用指南	60
2.4.1	简介	60
2.4.2	libyuv 扩展说明	60
2.5	SOPHGO JPEG 使用指南	75
2.5.1	简介	75
2.5.2	JPEG 数据结构说明	75
2.5.3	JPEG 接口说明	91
2.5.4	JPEG 测试用例说明	98
2.6	SOPHGO Video Decoder 使用指南	101
2.6.1	简介	101
2.6.2	VDEC 数据类型介绍	101

2.6.3	VDEC API 介绍 . . . . .	110
2.6.4	VDEC API 测试例程 . . . . .	117
2.7	SOPHGO Video Encoder 使用指南 . . . . .	118
2.7.1	简介 . . . . .	118
2.7.2	数据结构说明 . . . . .	118
2.7.3	API 扩展说明 . . . . .	125

## 发布记录

版本	发布日期	说明
V2.0.2	2019.11.15	第一版增加 OpenCV 私有 API、FFMPEG API，增强 libyuv API 接口介绍
V2.1.0	2020.07.06	第二版优化 FFMPEG 部分的章节结构，增加视频编码部分的内容介绍
V2.2.0	2020.08.26	第三版调整字体，优化版式，增加 bmx264 视频编码器的内容
V2.2.1	2021.02.09	更新 Opencv 新增接口的说明： bmcv::/av::/Mat:: 下新增接口



### 法律声明

版权所有 © 算能 2022. 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

### 注意

您购买的产品、服务或特性等应受算能商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，算能对本文档内容不做任何明示或默示的声明或保证。由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

## 技术支持

### 地址

北京市海淀区丰豪东路 9 号院中关村集成电路设计园 (ICPARK) 1 号楼

### 邮编

100094

### 网址

<https://www.sophgo.com/>

### 邮箱

[sales@sophgo.com](mailto:sales@sophgo.com)

### 电话

+86-10-57590723 +86-10-57590724

## SDK 发布记录

版本	发布日期	说明
v22.09.02	2022.09.02	第一次发布。
v22.10.01	2022.10.01	第二次发布。
v22.12.01	2022.12.01	2022 年 12 月第一次发布。
v23.03.01	2023.03.01	2023 年 3 月第一次发布。
v23.05.01	2023.05.01	2023 年 5 月第一次发布。
v23.07.01	2023.07.01	2023 年 7 月第一次发布。
v23.10.01	2023.10.01	2023 年 10 月第一次发布。

## 2.1 SOPHGO 多媒体框架介绍

### 2.1.1 简介

本文档所述多媒体框架的描述对象为算能的算丰 BM1686 产品系列，目前该产品系列仅包括 BM1686。其中 1) 本文中所有关于视频硬件编码的内容均只针对 BM1686 而言；2) 本文中提到的 Opencv 中的 bmcv 名字空间下的函数，仅针对 BM1686 版本产品而言。

本文档所述多媒体框架的覆盖范围包括 BM1686 产品系列中的视频解码 VPU 模块、视频编码 VPU 模块、图像编码 JPU 模块、图像解码 JPU 模块、图像处理模块 VPP。这些模块的功能封装到 FFMPEG 和 OPENCV 开源框架中，客户可以根据自己的开发习惯，选择 FFMPEG API 或者 OPENCV API。其中图像处理模块，我们还单独提供了算能自有的 BMCV API 底层接口，这部分接口有专门的文档介绍，可以参考《BMCV User Guide》，本文档不再详细介绍，仅介绍这三套 API 之间的层级关系及如何互相转换。

OPENCV，FFMPEG 和 BMCV 这三套 API 在功能上是子集的关系，但有少部分不能全部包含，下面的括号中进行了特别标注。

- 1) BMCV API 包含了所有能用硬件加速的图像处理加速接口（这里图像处理硬件加速，包括硬件图像处理 VPP 模块加速，以及借用其他硬件模块实现的图像处理功能）
- 2) FFMPEG API 包含了所有硬件加速的视频/图像编解码接口，所有软件支持的视频/图像编解码接口（即所有 FFMPEG 开源支持的格式），通过 bm\_scale filter 支持的部分硬件加速的图像处理接口（这部分图像处理接口，仅包括用硬件图像处理 VPP 模块加速的缩放、crop、padding、色彩转换功能）
- 3) OPENCV API 包含了所有 FFMPEG 支持的硬件及软件视频编解码接口（视频底层通过 FFMPEG 支持，这部分功能完全覆盖），硬件加速的 JPEG 编解码接口，软件支持的其他所有图像编解码接口（即所有 OPENCV 开源支持的图像格式），部分硬件加速的图像处理接

口（指用图像处理 VPP 模块加速的缩放、crop、padding、色彩转换功能），所有软件支持的 OPENCV 图像处理功能。

这三个框架中，BMCV 专注于图像处理功能，且能用 BM1686 硬件加速的部分；FFMPEG 框架强于图像和视频的编解码，几乎所有格式都可以支持，只是是否能用硬件加速的区别；OPENCV 框架强于图像处理，各种图像处理算法最初都先集成到 OPENCV 框架中，而视频编解码通过底层调用 FFMPEG 来实现。

因为 BMCV 仅提供了图像处理接口，因此 FFMPEG 或者 OPENCV 框架中，客户一般会选择一个作为主框架进行开发。这两个框架，从功能抽象上来说，OPENCV 的接口要更加简洁，一个接口就可以实现一次视频编解码操作；从性能上说，这两个的性能是完全一致的，几乎没有差别，在视频编解码上，OPENCV 只是对 FFMPEG 接口的一层封装；从灵活性上说，FFMPEG 的接口更加分离，可插入的操作粒度更细。最重要的，客户还是要根据自己对于某个框架的熟悉程度来做选择，只有深入了解，才能把框架用好。

这三个框架层级关系如图所示

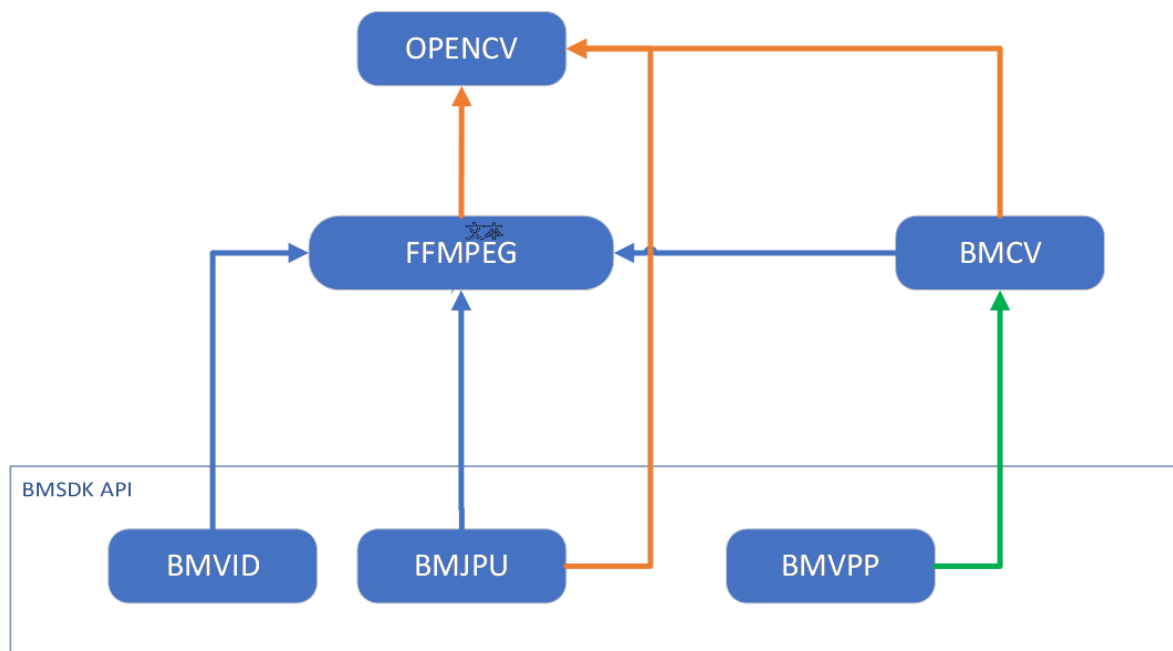


图 1 OPENCV/FFMPEG/BMCV 与 BMSDK 之间的层级调用关系

在很多应用场景下，需要用到某个框架下的特殊功能，因此在第 4 节中给出了三个框架之间灵活转换的方案。这种转换是不需要发生大量数据拷贝的，对性能几乎没有损失。



### 2.1.2 BM1686 硬件加速功能

本节给出了多媒体框架中硬件加速模块能支持的功能。其中硬件加速模块包括视频解码 VPU 模块，视频编码 VPU 模块，图像编解码 JPU 模块，图像处理 VPP 模块。

需要特别注意，这里只列出能够用硬件加速的能力，以及典型场景下的性能估计值。更详细的性能指标参考 BM1686 产品规格书。

#### 视频编解码

BM1686 产品支持 H264 (AVC), HEVC 视频格式的硬件解码加速，最高支持到 4K 视频的实时解码。支持 H264(AVC), HEVC 视频格式的硬件编码，最高支持到 HD(1080p) 视频的实时编码。

视频解码的速度与输入视频码流的格式有很大关系，不同复杂度的码流的解码速度有比较大的波动，比如码率、GOP 结构，分辨率等，都会影响到具体的测试结果。一般来说，针对视频监控应用场景，BM1686 产品单芯片可以支持到 32 路 HD 高清实时解码。

视频编码的速度与编码的配置参数有很大关系，不同的编码配置下，即使相同的视频内容，编码速度也不是完全相同的。一般来说，BM1686 产品单芯片最高可以支持到 2 路 HD 高清实时编码。

#### 图像编解码

BM1686 产品支持 JPEG baseline 格式的硬件编/解码加速。注意，仅支持 JPEG baseline 档次的硬件编解码加速，对于其他图片格式，包括 JPEG2000, BMP, PNG 以及 JPEG 标准的 progressive, lossless 等档次均自动采用软解支持。在 opencv 框架中，这种兼容支持对于客户是透明的，客户应用开发时无需特别处理。

图像硬件编解码的处理速度和图像的分辨率、图像色彩空间 (YUV420/422/444) 有比较大的关系，一般而言，对于 1920x1080 分辨率的图片，色彩空间为 YUV420 的，单芯片图像硬件编解码可以达到 600fps 左右。

#### 图像处理

BM1686 产品有专门的视频处理 VPP 单元对图像进行硬件加速处理。支持的图像操作有色彩转换、图像缩放、图像切割 crop、图像拼接 stitch 功能。最大支持到 4k 图像输入。对于 VPP 不支持的一些常用复杂图像处理功能，如线性变换  $ax+b$ ，直方图等，我们在 BMCV API 接口中，利用其他硬件单元做了特殊的加速处理。

### 2.1.3 硬件内存分类

在后续的讨论中，内存同步问题是应用调试中经常会遇到的，比较隐蔽的问题。我们通常统一用设备内存和系统内存来称呼这两类内存间的同步。

SOC 模式，是指用 BM1686 芯片中的处理器作为主控 CPU，BM1686 产品独立运行应用程序。典型的产品有 SE5、SM5-soc 模组。在这类模式下，采用 Linux 系统下的 ION 内存对设备内存进行管理。在 SOC 模式下，设备内存指 ION 分配的物理内存，系统内存其实是 cache，这里的命名只是为了和 PCI E 模式保持一致。从系统内存 (cache) 到设备内存，称为 Upload 上传 (实质是 cache flush)；从设备内存到系统内存 (cache)，称为 Download 下载 (实质是 cache invalidation)。在 SOC 模式下，设备内存和系统内存最终操作的其实是同一块物理内存，大部分时间，操作系统会自动对其进行同步，这也导致内存没有及时同步时的现象更加隐蔽和难以复现。

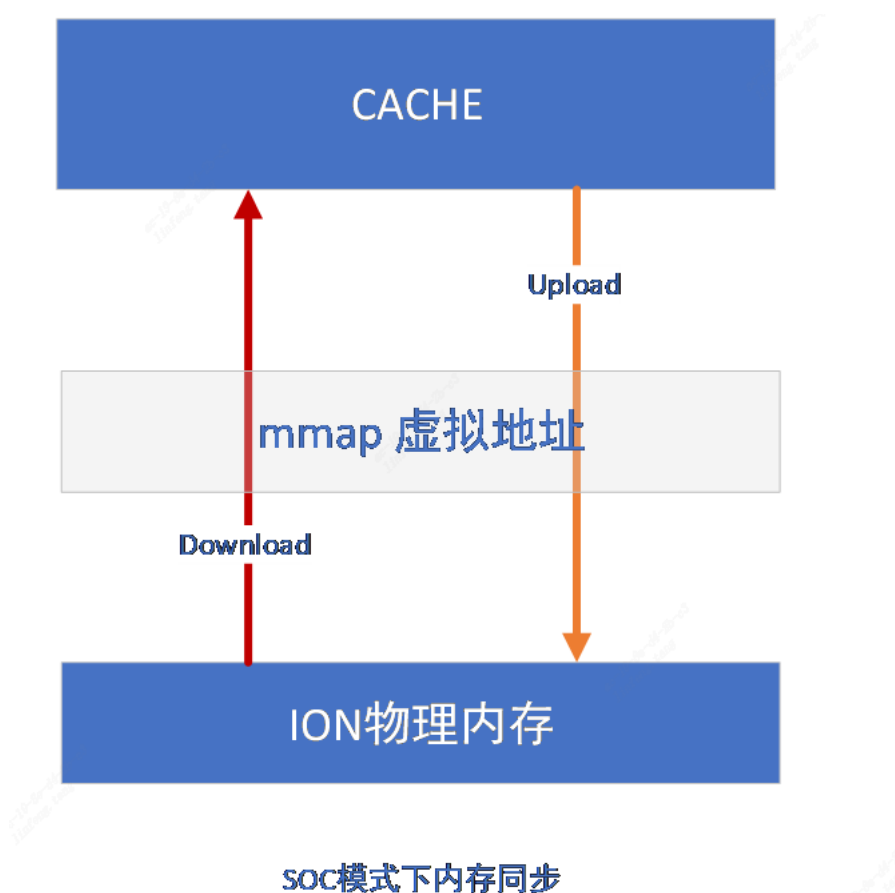


图 2 内存同步模型

FFMPEG 和 OPENCV 两个框架都提供了内存同步操作的函数。而 BMCV API 只面向设备内存操作，因此不存在内存同步的问题，在调用 BMCV API 的时候，需要将数据在设备内存准备好。

在 OPENCV 框架中，部分函数的形参中就提供了 update 的标志位，当标志位设置 true 的时候，函数内部会自动进行内存同步操作。这部分可以参考后续的第二章第 3 节的 API 介绍。用户也可以通过 `bmcv::downloadMat()` 和 `bmcv::uploadMat()` 两个函数，主动控制内存同步。

同步的基本原则是：a) opencv 原生函数中，yuv Mat 格式下设备内存中的数据永远是最新的，RGB Mat 格式下系统内存中的数据永远是最新的 b) 当 opencv 函数向 BMCV API 切换的时候，根据上一个原则，将最新数据同步到设备内存中；反之，从 BMCV API 向 opencv 函数切换的时候，在 RGB Mat 下将最新数据同步到系统内存中。c) 在不发生框架切换的时候，要尽量减少内存同步的操作。频繁的内存同步操作会明显降低性能。

在常规 FFMPEG 框架中，有两类称之为软（常规）和硬（hwaccel）的 codec API 和 filter API。这两套 API 的框架都可以支持 BM1686 的硬件视频编解码和硬件图像 filter，从这个角度上说，所谓的软解码和硬解码在底层性能上是完全一样的，只是在使用习惯上的区别。软 codec/filter API 的使用方式和通常 ffmpeg 内置 codec 完全一致，硬 codec/filter API 要用-hwaccel 来指定使能 bmcodec 专用硬件设备。当在软 codec API 和 filter API 中，通过 av\_dict\_set 传入标志参数 “is\_dma\_buffer” 或者 “zero\_copy”，来控制内部 codec 或 filter 是否将设备内存数据同步到系统内存中，具体参数使用可以用 ffmpeg -h 来查看。当后续直接衔接硬件处理的时候，通常不需要将设备内存数据同步到系统内存中。

在 hwaccel codec API 和 filter API 中，内存默认只有设备内存，没有分配系统内存。如果需要内存同步，则要通过 hwupload 和 hwdownload filter 来完成。

综上所述，OPENCV 和 FFMPEG 框架都对内存同步提供了支持，应用可以根据自己的使用习惯选择相应的框架，对数据同步进行精准控制。BMCV API 则始终工作在设备内存上。

### 2.1.4 框架之间转换

在应用开发中，总会碰到一些情况下，某个框架无法完全满足设计需求。这时就需要在各种框架之间快速切换。BM1686 多媒体框架对其提供了支持，可以满足这种需求，并且这种切换是不发生数据拷贝的，对于性能几乎没有影响。

#### FFMPEG 和 OPENCV 转换

FFMPEG 和 OPENCV 之间的转换，主要是数据格式 AVFrame 和 cv::Mat 之间的格式转换。

当需要 FFMPEG 和 OPENCV 配合解决的时候，推荐使用常规非 HWAaccel API 的通路，目前 OPENCV 内部采用是这种方式，验证比较完备。

FFMPEG AVFrame 转到 OPENCV Mat 格式如下。

1. AVFrame \* picture;
2. 中间经过 ffmpeg API 的一系列处理，比如 avcodec\_decode\_video2 或者 avcodec\_receive\_frame，然后将得到的结果转成 Mat
4. card\_id 为进行 ffmpeg 硬件加速解码的设备序号，在常规 codec API 中，通过 av\_dict\_set 的 sophon\_idx 指定，或者 hwaccel API 中，在 hwaccel 设备初始化的时候指定，soc 模式下默认为 0
5. cv::Mat ocv\_frame(picture, card\_id);
6. 或可以通过分步方式进行格式转换
7. cv::Mat ocv\_frame;

8. `ocv_frame.create(picture, card_id);`
9. 然后可以用 `ocv_frame` 进行 `opencv` 的操作, 此时 `ocv_frame` 格式为 `BM1686` 扩展的 `yuv mat` 类型, 如果后续想转成 `opencv` 标准的 `bgr mat` 格式, 可以按下列操作。
10. 注意: 这里就有内存同步的操作, 如果没有设置, `ffmpeg` 默认是在设备内存中的, 如果 `update=false`, 那么转成 `bgr` 的数据也一直在设备内存中, 系统内存中为无效数据, 如果 `update=true`, 则设备内存同步到系统内存中。如果后续还是硬件加速处理的话, 可以 `update=false`, 这样可以提高效率, 当需要用到系统内存数据的时候, 显式调用 `bmcv::downloadMat()` 来同步即可。
11. `cv::Mat bgr_mat;`
12. `cv::bmcv::toMAT(ocv_frame, bgr_mat, update);`
13. 最后 `AVFrame *picture` 会被 `Mat ocv_frame` 释放, 因此不需要对 `picture` 进行 `av_frame_free()` 操作。如果希望外部调用 `av_frame_free` 来释放 `picture`, 则可以加上 `card_id = card_id | BM_MAKEFLAG(UMatData::AVFRAME_ATTACHED, 0, 0)`, 该标准表明 `AVFrame` 的创建和释放由外部管理
14. `ocv_frame.release();`
15. `picture = nullptr;`

`OPENCV Mat` 转成 `FFMPEG AVFrame` 的情况比较少见, 因为几乎所有需要的 `FFMPEG` 操作都在 `opencv` 中有对应的封装接口。比如: `ffmpeg` 解码在 `opencv` 有 `videoCapture` 类, `ffmpeg` 编码在 `opencv` 中有 `videoWriter` 类, `ffmpeg` 的 `filter` 操作对应的图像处理在 `opencv` 中有 `bmcv` 名字空间下的接口以及丰富的原生图像处理函数。

一般来说, `opencv Mat` 转成 `FFMPEG AVFrame`, 指的是 `yuv Mat`。在这种情况下, 可以按下进行转换。

1. 创建 `yuv Mat`, 如果 `yuv Mat` 已经存在, 可以忽略此步。`card_id` 为 `BM1686` 设备序号, `soc` 模式下默认为 0
2. `AVFrame * f = cv::av::create(height, width, AV_PIX_FMT_YUV420P, NULL, 0, -1, NULL, NULL, AVCOL_SPC_BT709, AVCOL_RANGE_MPEG, card_id);`
3. `cv::Mat image(f, card_id);`
4. do something in `opencv`
5. `AVFrame * frame = image.u->frame;`
6. call `FFMPEG API`
7. 注意: 在 `ffmpeg` 调用完成前, 必须保证 `Mat image` 没有被释放, 否则 `AVFrame` 会和 `Mat image` 一起释放。如果需要将两个的声明周期分离开来, 则上面的 `image` 声明要改成如下格式。
8. `cv::Mat image(f, card_id | BM_MAKEFLAG(UMatData::AVFRAME_ATTACHED, 0, 0));`
9. 这样 `Mat` 就不会接管 `AVFrame` 的内存释放工作

## FFMPEG 和 BMCV API 转换

FFMPEG 经常需要和 BMCV API 搭配使用，因此 FFMPEG 和 BMCV 之间的转换是比较频繁的。为此我们专门给了一个例子 `ff_bmcv_transcode`，该例子可以在 `bmnnsdk2` 发布包里找到。

`ff_bmcv_transcode` 例子演示了用 `ffmpeg` 解码，将解码结果转换到 BMCV 下进行处理，然后再转换回到 `ffmpeg` 进行编码的过程。FFMPEG 和 BMCV 之间的互相转换可以参考 `ff_avframe_convert.cpp` 文件中的 `avframe_to_bm_image()` 和 `bm_image_to_avframe()` 函数。

## OPENCV 和 BMCV API 转换

OPENCV 和 BMCV API 之间的转换，专门在 `opencv` 扩展的 `bmcv` 名字空间下提供了专门的转换函数。

OPENCV Mat 转换到 BMCV `bm_image` 格式。

1. `cv::Mat m(height, width, CV_8UC3, card_id);`
2. `opencv` 操作
3. `bm_image bmcv_image;`
4. 这里 `update` 用来控制内存同步，是否需要内存同步取决于前面的 `opencv` 操作，如果前面的操作都是用硬件加速完成，设备内存中就是最新数据，就没必要进行内存同步，如果前面的操作调用了 `opencv` 函数，没有使用硬件加速（后续 `opencv` 章节 6.2 中提到了哪些函数采用了硬件加速），对于 `bgr mat` 格式就需要做内存同步。
5. 也可以在调用下面函数之前，显式的调用 `cv::bmcv::uploadMat(m)` 来实现内存同步
6. `cv::bmcv::toBMI(m, &bmcv_image, update);`
7. 使用 `bmcv_image` 就可以进行 `bmcv api` 调用，调用期间注意保证 `Mat m` 不能被释放，因为 `bmcv_image` 使用的是 `Mat m` 中分配的内存空间。handle 可以通过 `bm_image_get_handle()` 获得
8. 释放：必须调用此函数，因为在 `toBMI` 中 `create` 了 `bm_image`，否则会有内存泄漏
9. `bm_image_destroy(bmcv_image);`
10. `m.release();`

由 BMCV `bm_image` 格式转换到 OPENCV Mat 有两种方式，一种是会发生数据拷贝，这样 `bm_image` 和 `Mat` 之间相互独立，可以分别释放，但是有性能损失；一种是直接引用 `bm_image` 内存，性能没有任何损失。

1. `bm_image bmcv_image;`
2. 调用 `bmcv API` 给 `bmcv_image` 分配内存空间，并进行操作
3. `Mat m_copy, m_nocopy;`
4. 下面接口将发生内存数据拷贝，转换成标准 `bgr mat` 格式。

5. update 控制内存同步，也可以在调用完这个函数后用 `bmcv::downloadMat()` 来控制内存同步
6. `csc_type` 是控制颜色转换系数矩阵，控制不同 yuv 色彩空间转换到 bgr
7. `cv::bmcv::toMAT(&bmcv_image, m_copy, update, csc_type);`
8. 下面接口接口将直接引用 `bm_image` 内存 (nocopy 标志位 true), update 仍然按照之前的描述，
9. 选择是否同步内存。在后续 opencv 操作中，必须保证 `bmcv_image` 没有释放，因为 mat 的内存
10. 直接引用自 `bm_image` `cv::bmcv::toMAT(&bmcv_image, &m_nocopy, AVCOL_SPC_BT709, AVCOL_RANGE_MPEG, NULL, -1, update, true);`
11. 进行 opencv

## 2.2 SOPHGO OpenCV 使用指南

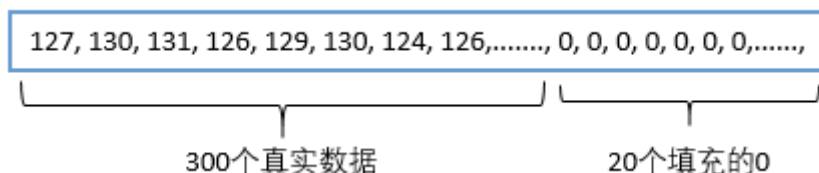
### 2.2.1 OpenCV 简介

BM1686 系列芯片中的多媒体、BMCV 和 NPU 硬件模块可以加速对图片和视频的处理：

- 1) 多媒体模块：硬件加速 JPEG 编码解码和 Video 编解码操作。
- 2) BMCV 模块：硬件加速对图片的 `resize`、`color conversion`、`crop`、`split`、`linear transform`、`nms`、`sort` 等操作。
- 3) NPU 模块：硬件加速对图片的 `split`、`rgb2gray`、`mean`、`scale`、`int8tofloat32` 操作。

为了方便客户使用芯片上的硬件模块加速图片和视频的处理，提升应用 OpenCV 软件性能，算能修改了 OpenCV 库，在其内部调用硬件模块进行 Image 和 Video 相关的处理。

算能当前 OpenCV 的版本为 4.1.0，除了以下几个算能自有的 API 以外，其它的所有 API 与 OpenCV API 都是一致的。



在 SOC 模式下，由于填充了多余的 0 值，Mat 对象中存储数据的 `data` 变量不能直接传递给 BMRuntime 库的 API 做推理，否则会降低模型的准确率。请在最后一次 BMCV 做变换的时候，将 `stride` 设置为非对齐方式，多余的 0 会被自动清除掉。

### 2.2.2 数据结构扩展说明

OpenCV 内置标准处理的色彩空间为 BGR 格式，但是很多情况下，对于视频、图片源，直接在 YUV 色彩空间上处理，可以节省带宽和避免不必要的 YUV 和 RGB 之间的互相转换。因此 SOPHGO Opencv 对于 Mat 类进行了扩展。

- 1) 在 Mat.UMatData 中, 引入了 AVFrame 成员, 扩展支持各种 YUV 格式。其中 AVFrame 的格式定义与 FFMPEG 中的定义兼容
- 2) 在 Mat.UMatData 中增加了 fd, addr (soc 模式下) 的定义, 分别表示对应的内存管理句柄和物理内存地址
- 3) 在 Mat 类中增加了 fromhardware 变量, 标识当前的视频、图片解码是由硬件或是软件计算完成的, 开发者在程序开发过程中无需考虑该变量的值。

### 2.2.3 API 扩展说明

**bool VideoCapture::get\_resampler(int \*den, int \*num)**

函数原型	bool VideoCapture::get_resampler(int *den, int *num)
功能	取视频的采样速率。如 den=5, num=3 表示每 5 帧中有 2 帧被丢弃
输入参数	int *den - 采样速率的分母 int *num - 采样速率的分子
输出参数	无
返回值	true - 执行成功 false - 执行失败
说明	此接口将废弃。推荐用 double VideoCapture::get(CAP_PROP_OUTPUT_SRC) 接口。

**bool VideoCapture::set\_resampler(int den, int num)**

函数原型	bool VideoCapture::set_resampler(int den, int num)
功能	置视频的采样速率。如 den=5, num=3, 表示每 5 帧中有 2 帧被丢弃。
输入参数	int den - 采样速率的分母 int num - 采样速率的分子
输出参数	无
返回值	true - 执行成功 false - 执行失败
说明	此接口将废弃。推荐用 bool VideoCapture::set(CAP_PROP_OUTPUT_SRC, double resampler) 接口。



**double VideoCapture::get(CAP\_PROP\_TIMESTAMP)**

函数原型	double VideoCapture::get(CAP_PROP_TIMESTAMP)
功能	提供当前图片的时间戳，时间基数取决于在流中给出的时间基数
输入参数	CAP_PROP_TIMESTAMP – 特定的枚举类型指示获取时间戳，此类型由 Sophgo 定义
输出参数	无
返回值	在使用前先将返回值转成 unsigned int64 数据类型 0x8000000000000000L-No AV PTS value other-AV PTS value

**double VideoCapture::get(CAP\_PROP\_STATUS)**

函数原型	double VideoCapture::get(CAP_PROP_STATUS)
功能	该函数提供了一个接口，用于检查视频抓取的内部运行状态
输入参数	CAP_PROP_STATUS – 枚举类型，此类型由 Sophgo 定义
输出参数	无
返回值	在使用返回值前请将转换成 int 类型 0 视频抓取停止，暂停或者其他无法运行的状态 1 视频抓取正在进行 2 视频抓取结束

**bool VideoCapture::set(CAP\_PROP\_OUTPUT\_SRC, double resampler)**

函数原型	double VideoCapture::get(CAP_PROP_OUTPUT_SRC, double resampler)
功能	设置 YUV 视频的采样速率。如 resampler 为 0.4，表示每 5 帧中保留 2 帧，有 3 帧被丢弃
输入参数	CAP_PROP_OUTPUT_SRC – 枚举类型，此类型由 Sophgo 定义 double resampler – 采样速率
输出参数	无
返回值	true 执行成功 false 执行失败



**double VideoCapture::get(CAP\_PROP\_OUTPUT\_SRC)**

函数原型	double VideoCapture::get(CAP_PROP_OUTPUT_SRC)
功能	取视频的采样速率。
输入参数	CAP_PROP_OUTPUT_SRC - 特定的枚举类型，指视频输出，此类型由 SOPHGO 定义
输出参数	无
返回值	采样率数值

**bool VideoCapture::set(CAP\_PROP\_OUTPUT\_YUV, double enable)**

函数原型	bool VideoCapture::set(CAP_PROP_OUTPUT_YUV, double enable)
功能	开或者关闭 YUV 格式的 frame 输出。BM1686 系列下 YUV 格式为 I420
输入参数	CAP_PROP_OUTPUT_YUV - 特定的枚举类型，指 YUV 格式的视频 frame 输出，此类型由 SOPHGO 定义； double enable - 操作码，1 表示打开，0 表示关闭
输出参数	无
返回值	true: 执行成功 false: 执行失败

**double VideoCapture::get(CAP\_PROP\_OUTPUT\_YUV)**

函数原型	double VideoCapture::get(CAP_PROP_OUTPUT_YUV)
功能	取 YUV 视频 frame 输出的状态。
输入参数	CAP_PROP_OUTPUT_YUV - 特定的枚举类型，指 YUV 格式的视频 frame 输出，此类型由 SOPHGO 定义。
输出参数	无
返回值	YUV 视频 frame 输出的状态。1 表示打开，0 表示关闭。

**bm\_status\_t bmcv::toBMI(Mat &m, bm\_image \*image, bool update = true)**

函数原型	bm_status_t bmcv::toBMI(Mat &m, bm_image *image, bool date = true)
功能	OpenCV Mat 对象转换成 BMCV 接口中对应格式的 bm_image 数据对象，本接口直接引用 Mat 的数据指针，不会发生 copy 操作。本接口仅支持 1N 模式
输入参数	Mat& m - Mat 对象，可以为扩展 YUV 格式或者标准 OpenCV BGR 格式； bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	bm_image *image - 对应格式的 BMCV bm_image 数据对象
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败
说明	目前支持压缩格式、Gray、NV12、NV16, YUV444P、YUV422P、YUV420P、BGR separate、BGR packed、CV_8UC1 的格式转换

**bm\_status\_t bmcv::toBMI(Mat &m, Mat &m1, Mat &m2, Mat &m3, bm\_image \*image, bool update = true)**

函数原型	bm_status_t bmcv::toBMI(Mat &m, Mat &m1, Mat &m2, Mat &m3, bm_image *image, bool update = true)
功能	OpenCV Mat 对象转换成 BMCV 接口中对应格式的 bm_image 数据对象，本接口直接引用 Mat 的数据指针，不发生 copy 操作。本接口针对 BMCV 的 4N 模式。要求所有 Mat 的输入图像格式一致, 仅对 BM1686 有效
输入参数	Mat &m - 4N 中的第 1 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 Mat &m1 - 4N 中的第 2 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 Mat &m2 - 4N 中的第 3 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 Mat &m3 - 4N 中的第 4 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	bm_image *image - 对应格式的 BMCV bm_image 数据对象，其中包含 4 个图像数据
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败
说明	目前支持压缩格式、Gray、NV12、NV16, YUV444P、YUV422P、YUV420P、BGR separate、BGR packed、CV_8UC1 的格式转换

**bm\_status\_t bmcv::toMAT(Mat &in, Mat &m0, bool update=true)**

函数原型	bm_status_t bmcv::toMAT(Mat &in, Mat &m0, bool update = true)
功能	输入的 MAT 对象, 可以为各种 YUV 或 BGR 格式, 转换成 BGR packet 格式的 MAT 对象输出
输入参数	Mat &in - 输入的 MAT 对象, 可以为各种 YUV 格式或者 BGR 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	Mat &m0 - 输出的 MAT 对象, 转成标准 OpenCV 的 BGR 格式
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败
说明	目前支持压缩格式、Gray、NV12、NV16, YUV444P、YUV422P、YUV420P、BGR separate、BGR packed、CV_8UC1 到 BGR packed 格式转换。在 YUV 格式下, 会自动根据 AVFrame 结构体中 color_space,color_range 信息选择正确的色彩转换矩阵。

`bm_status_t toMAT(bm_image *image, Mat &m, int color_space, int color_range, void* vaddr = NULL, int fd0 = -1, bool update = true, bool nocopy = true)`

函数原型	<code>bm_status_t bmcv::toMAT(bm_image *image, Mat &amp;m, int color_space, int color_range, void* vaddr=NULL, int fd0=-1, bool update=true, bool nocopy=true)</code>
功能	输入的 <code>bm_image</code> 对象, 当 <code>nocopy</code> 为 <code>true</code> 时, 直接复用设备内存转成 <code>Mat</code> 格式, 当 <code>nocopy</code> 为 <code>false</code> 时, 行为类似 3.13toMAT 接口, 1N 模式。
输入参数	<p><code>bm_image *image</code> - 输入的 <code>bm_image</code> 对象, 可以为各种 YUV 格式或者 BGR 格式;</p> <p><code>Int color_space</code> - 输入 <code>image</code> 的色彩空间, 可以为 <code>AVCOL_SPC_BT709</code> 或 <code>AVCOL_SPC_BT470</code>, 详见 <code>FFMPEG pixfmt.h</code> 定义;</p> <p><code>Int color_range</code> - 输入 <code>image</code> 的色彩动态范围, 可以为 <code>AVCOL_RANGE_MPEG</code> 或 <code>AVCOL_RANGE_JPEG</code>, 详见 <code>FFMPEG pixfmt.h</code> 定义;</p> <p><code>Void* vaddr</code> - 输出 <code>Mat</code> 的系统虚拟内存指针, 如果已分配, 输出 <code>Mat</code> 直接使用该内存作为 <code>Mat</code> 的系统内存。如果为 <code>NULL</code>, 则 <code>Mat</code> 内部自动分配;</p> <p><code>Int fd0</code> - 输出 <code>Mat</code> 的物理内存句柄, 如果为负, 则使用 <code>bm_image</code> 内的设备内存句柄, 否则使用 <code>fd0</code> 给定的句柄来 <code>mmap</code> 设备内存;</p> <p><code>bool update</code> - 是否需要同步 <code>cache</code> 或内存。如果为 <code>true</code>, 则转换完成后同步 <code>cache</code></p> <p><code>bool nocopy</code> - 如果是 <code>true</code>, 则直接引用 <code>bm_image</code> 的设备内存, 如果为 <code>false</code>, 则转换成标准 BGR <code>Mat</code> 格式</p>
输出参数	<code>Mat &amp;m</code> - 输出的 <code>MAT</code> 对象, 当 <code>nocopy</code> 为 <code>true</code> 时, 输出标准 BGR 格式或扩展的 YUV 格式的 <code>Mat</code> ; 当 <code>nocopy</code> 为 <code>false</code> 时, 转成标准 OpenCV 的 BGR 格式。
返回值	<code>BM_SUCCESS(0)</code> : 执行成功其他: 执行失败
说明	<ol style="list-style-type: none"> <li>1.no copy 方式只支持 1N 模式, 4N 模式因为内存排列方式, 不能支持引用</li> <li>2. 在 <code>nocopy</code> 为 <code>false</code> 的情况下, 会自动根据参数 <code>colorspace,color_range</code> 信息选择正确的色彩转换矩阵进行色彩转换。</li> <li>3. 如果系统内存 <code>vaddr</code> 来自于外部, 那么外部需要来管理这个内存的释放, <code>Mat</code> 释放的时候不会释放该内存</li> </ol>

**bm\_status\_t bmcv::toMAT(bm\_image \*image, Mat &m0, bool update = true, csc\_type\_t csc = CSC\_MAX\_ENUM)**

函数原型	bm_status_t bmcv::toMAT(bm_image *image, Mat &m0, bool update=true, csc_type_t csc=CSC_MAX_ENUM)
功能	输入的 bm_image 对象, 可以为各种 YUV 或 BGR 格式, 转换成 BGR 格式的 MAT 对象输出, 1N 模式
输入参数	bm_image *image - 输入的 bm_image 对象, 可以为各种 YUV 格式或者 BGR 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache csc_type_t csc - 色彩转换类型, 仅当输入 bm_image 为 YUV 格式时需要 csc 转换, 默认 type 为 YCbCr2RGB_BT601
输出参数	Mat &m0 - 输出的 MAT 对象, 转成标准 OpenCV 的 BGR 格式
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败

**bm\_status\_t bmcv::toMAT(bm\_image \*image, Mat &m0, Mat &m1, Mat &m2, Mat &m3, bool update=true, csc\_type\_t csc=CSC\_MAX\_ENUM)**

函数原型	bm_status_t bmcv::toMAT(bm_image *image, Mat &m0, Mat &m1, Mat &m2, Mat &m3, bool update=true, csc_type_t csc=CSC_MAX_ENUM)
功能	输入的 bm_image 对象, 可以为各种 YUV 或 BGR 格式, 转换成 BGR 格式的 MAT 对象输出, 4N 模式, 仅在 BM1686 下有效
输入参数	bm_image *image - 输入的 4N 模式下的 bm_image 对象, 可以为各种 YUV 格式或者 BGR 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache csc_type_t csc - 色彩转换类型, 仅当输入 bm_image 为 YUV 格式时需要 csc 转换, 默认 type 为 YCbCr2RGB_BT601
输出参数	Mat &m0 - 输出的第一个 MAT 对象, 转成标准 OpenCV 的 BGR 格式; Mat &m1 - 输出的第二个 MAT 对象, 转成标准 OpenCV 的 BGR 格式; Mat &m2 - 输出的第三个 MAT 对象, 转成标准 OpenCV 的 BGR 格式; Mat &m3 - 输出的第四个 MAT 对象, 转成标准 OpenCV 的 BGR 格式
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败

**bm\_status\_t bmcv::resize(Mat &m, Mat &out, bool update = true, int interpolation=BMCV\_INTER\_NEAREST)**

函数原型	bm_status_t bmcv::resize(Mat &m, Mat &out, bool update = true, int interpolation = BMCV_INTER_NEAREST)
功能	输入的 MAT 对象，缩放到输出 Mat 给定的大小，输出格式为输出 Mat 指定的色彩空间，因为 MAT 支持扩展的 YUV 格式，因此本接口支持的色彩空间并不仅限于 BGR packed。
输入参数	Mat &m - 输入的 Mat 对象，可以为标准 BGR packed 格式或者扩展 YUV 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache int interpolation - 缩放算法，可为 NEAREST 或者 LINEAR 算法
输出参数	Mat &out - 输出的缩放后的 Mat 对象
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败
说明	支持 Gray、YUV444P、YUV420P、BGR/RGB separate、BGR/RGB packed、ARGB packed 格式缩放

**bm\_status\_t bmcv::convert(Mat &m, Mat &out, bool update=true)**

函数原型	bm_status_t bmcv::convert(Mat &m, Mat &out, bool update = true)
功能	实现两个 mat 之间的色彩转换，它与 toMat 接口的区别在于 toMat 只能实现各种色彩格式到 BGR packed 的色彩转换，而本接口可以支持 BGR packed 或者 YUV 格式到 BGR packed 或 YUV 之间的转换。
输入参数	Mat &m - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR packed 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	Mat &out - 输出的色彩转换后的 Mat 对象，可以为 BGR packed 或者 YUV 格式。
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败

```
bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, std::vector<Size> &vsz,
std::vector<Mat> &out, bool update= true, csc_type_t csc=CSC_YCbCr2RGB_BT601,
csc_matrix_t *matrix = nullptr, bmcv_resize_algorithm algorithm= BMCV_INTER_LINEAR)
```

函数原型	bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, std::vector<Size> &vsz, std::vector<Mat> &out, bool update = true, csc_type_t csc=CSC_YCbCr2RGB_BT601, csc_matrix_t *matrix=nullptr, bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR)
功能	接口采用内置的 VPP 硬件加速单元，集 crop,resize 和 csc 于一体。按给定的多个 rect 框，给定的多个缩放 size，将输入的 Mat 对象，输出到多个 Mat 对象中，输出为 OpenCV 标准的 BGR pack 格式或扩展 YUV 格式
输入参数	Mat &m - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR packed 格式; std::vector<Rect> &vrt - 多个 rect 框，输入 Mat 中的 ROI 区域。矩形框个数和 resize 个数应该相同; std::vector<Size> &vsz - 多个 resize 大小，与 vrt 的矩形框一一对应; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache csc_type_t csc -色彩转换矩阵，可以根据颜色空间指定合适的色彩转换矩阵; csc_matrix_t *matrix -当色彩转换矩阵不在列表中时，可以给出外置的用户自定义的转换矩阵; bmcv_resize_algorithm algorithm -缩放算法，可以为 Nearest 或者 Linear 算法
输出参数	std::vector<Mat> &out - 输出的缩放、crop 以及色彩转换后的标准 BGR pack 格式或 YUV 格式的 Mat 对象。
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败
说明	接口可以将 resize,crop,csc 三种操作在一步之内完成，效率最高。在可能的情况下，要尽可能的使用该接口提高效率

```
bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, bm_image *out, bool update= true)
```

函数原型	bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, bm_image *out, bool update= true)
功能	接口采用内置的 VPP 硬件加速单元，集 crop,resize 和 csc 于一体。按给定的多个 rect 框，按照多个 bm_image 中指定的 size，将输入的 Mat 对象，输出到多个 bm_image 对象中，输出格式由 bm_image 初始化值决定。注意，bm_image 必须由调用者初始化好，并且个数和 vrt 一一对应。
输入参数	Mat &m - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR packed 格式; std::vector<Rect> &vrt - 多个 rect 框，输入 Mat 中的 ROI 区域。矩形框个数和 resize 个数应该相同; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	bm_image *out - 输出的缩放、crop 以及色彩转换后的 bm_image 对象，输出色彩格式由 bm_image 初始化值决定。同时该 bmimage 参数包含的初始化的 size、色彩信息也作为输入信息，用于处理。
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败

```
void bmcv::uploadMat(Mat &mat)
```

函数原型	void bmcv::uploadMat(Mat &mat)
功能	cache 同步或者设备内存同步接口。当执行此函数时，cache 中内容会 flush 到实际内存中（SOC 模式）
输入参数	Mat &mat - 输入的需要内存同步的 mat 对象
输出参数	无
返回值	无
说明	合理调用本接口，可以有效控制内存同步的次数，仅在需要的时候调用。

```
void bmcv::downloadMat(Mat &mat)
```

函数原型	void bmcv::downloadMat(Mat &mat)
功能	cache 同步或者设备内存同步接口。当执行此函数时，cache 中内容会 invalidate（SOC 模式）
输入参数	Mat &mat - 输入的需要内存同步的 mat 对象
输出参数	无
返回值	无
说明	合理调用本接口，可以有效控制内存同步的次数，仅在需要的时候调用。



`bm_status_t bmcv::stitch(std::vector<Mat> &in, std::vector<Rect>& src, std::vector<Rect>& drt, Mat &out, bool update = true, bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR)`

函数原型	<code>bm_status_t bmcv::stitch(std::vector&lt;Mat&gt; &amp;in, std::vector&lt;Rect&gt; &amp;src, std::vector&lt;Rect&gt; &amp;drt, Mat &amp;out, bool update=true, bmcv_resize_alogrithm algorithm=BMCV_INTER_LINEAR)</code>
功能	图像拼接，将输入的多个 Mat 按照按照给定的位置缩放并拼接到一个 Mat 中
输入参数	<code>std::vector&lt;Mat&gt; &amp;in</code> - 多个输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR pack 格式; <code>std::vector&lt;Rect&gt; &amp;src</code> - 对应每个 Mat 对象的显示内容框; <code>std::vector&lt;Rect&gt; &amp;drt</code> - 对应每个显示内容在目标 Mat 中的显示位置; <code>bool update</code> - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache <code>bmcv_resize_algorithm algorithm</code> - 缩放算法, 可以为 Nearest 或者 Linear 算法
输出参数	<code>Mat &amp;out</code> - 输出拼接后的 Mat 对象，可以为 BGR packed 或者 YUV 格式
返回值	BM_SUCCESS(0): 执行成功其他: 执行失败
说明	

`void bmcv::print(Mat &m, bool dump = false)`

函数原型	<code>void bmcv::print(Mat &amp;m, bool dump = false)</code>
功能	调试接口，打印输入 Mat 对象的色彩空间，宽高以及数据。
输入参数	<code>Mat &amp;m</code> - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR-packed 格式; <code>bool dump</code> - true 的时候打印 Mat 中的数据值，默认不打印。如果为 true, 则会在当前目录下生成 <code>mat_dump.bin</code> 文件
输出参数	无
返回值	无
说明	当前支持 dump OpenCV 标准 BGRpacked 或者 CV_8UC1 数据, 以及扩展的 NV12, NV16, YUV420P, YUV422P, GRAY, YUV444P 和 BGRSeparate 格式数据

**void bmcv::print(bm\_image \*image, bool dump)**

函数原型	void bmcv::print(bm_image *image, bool dump)
功能	调试接口，打印输入 bm_image 对象的色彩空间，宽高以及数据。
输入参数	bm_image *image - 输入的 bm_image 对象; bool dump - true 的时候打印 Mat 中的数据值，默认不打印，如果为 true，则会在当前目录下生成 BMI- “宽” x” 高” .bin 文件
输出参数	无
返回值	无
说明	当前支持 dump BGR packed,NV12,NV16,YUV420P,YUV422P,GRAY,YUV444P 和 BGR Separate 格式的 bm_image 数据

**void bmcv::dumpMat(Mat &image, const String &fname)**

函数原型	void bmcv::dumpMat(Mat &image, const String &fname)
功能	调试接口，专门 dumpMat 的数据到指定命名的文件。功能同 3.23 的 dump 为 true 时的功能。
输入参数	Mat &image - 输入的 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; const String &fname -dump 的输出文件名
输出参数	无
返回值	无
说明	当前支持 dump OpenCV 标准 BGR packed 或者 CV_8UC1 数据，以及扩展的 NV12,NV16,YUV420P,YUV422P,GRAY,YUV444P 和 BGR Separate 格式数据

**void bmcv::dumpBMImage(bm\_image \*image, const String &fname)**

函数原型	void bmcv::dumpBMImage(bm_image *image, const String &fname)
功能	调试接口，专门 dump bm_image 的数据到指定命名的文件。功能同 3.25 的 dump 为 true 时的功能。
输入参数	bm_image *image - 输入的 bm_image 对象; const String &fname -dump 的输出文件名
输出参数	无
返回值	无
说明	当前支持 dump BGR packed, NV12,NV16,YUV420P,YUV422P,GRAY,YUV444P 和 BGR Separate 格式的 bm_image 数据

**bool Mat::avOK()**

函数原型	bool Mat::avOK()
功能	判断当前 Mat 是否为扩展的 YUV 格式
输入参数	无
输出参数	无
返回值	true –表示当前 Mat 为扩展的 YUV 格式 false –表示当前 Mat 为标准 OpenCV 格式
说明	接口和接口 3.21 3.22 downloadMat、uploadMat 结合起来，可以有效地管理内存同步。 一般 avOK 为 true 的 Mat，物理内存是最新的，而 avOK 为 false 的 Mat，其 cache 或者 host 内存中的数据是最新的。可以根据这个信息，决定是调用 uploadMat 还是 downloadMat。 如果一直在设备内存中通过硬件加速单元工作，则可以省略内存同步，仅在需要交换到系统内存中时再调用 downloadMat。

**int Mat::avCols()**

函数原型	int Mat::avCols()
功能	获取 YUV 扩展格式的 Y 的宽
输入参数	无
输出参数	无
返回值	返回扩展的 YUV 格式的 Y 的宽，如果为标准 OpenCV Mat 格式，返回 0

**int Mat::avRows()**

函数原型	int Mat::avRows()
功能	获取 YUV 扩展格式的 Y 的高
输入参数	无
输出参数	无
返回值	返回扩展的 YUV 格式的 Y 的高，如果为标准 OpenCV Mat 格式，返回 0

**int Mat::avFormat()**

函数原型	int Mat::avFormat()
功能	获取 YUV 格式信息
输入参数	无
输出参数	无
返回值	返回扩展的 YUV 格式信息，如果为标准 OpenCV Mat 格式，返回 0

**int Mat::avAddr(int idx)**

函数原型	int Mat::avAddr(int idx)
功能	获取 YUV 各分量的物理地址
输入参数	int idx –指定 YUV plane 的序号
输出参数	无
返回值	返回指定的 plane 的物理首地址，如果为标准 OpenCV Mat 格式，返回 0

**int Mat::avStep(int idx)**

函数原型	int Mat::avStep(int idx)
功能	获取 YUV 格式中指定 plane 的 line size
输入参数	int idx –指定 YUV plane 的序号
输出参数	无
返回值	指定的 plane 的 line size，如果为标准 OpenCV Mat 格式，返回 0

```
AVFrame* av::create(int height, int width, int color_format, void *data, long addr, int fd, int*
plane_stride, int* plane_size, int color_space = AVCOL_SPC_BT709, int color_range = AV-
COL_RANGE_MPEG, int id = 0)
```

函数原型	AVFrame* av::create(int height, int width, int clor_format, void *data, long addr, int fd, int* plane_stride, int* plane_size, int color_space = AVCOL_SPC_BT709, int color_range = AVCOL_RANGE_MPEG, int id = 0)
功能	AVFrame 的创建接口，允许外部创建系统内存和物理内存，创建的格式与 FFMPEG 下的 AVFrame 定义兼容
输入参数	<p>int height –创建图像数据的高;</p> <p>int width –创建图像数据的宽;</p> <p>int color_format –创建图像数据的格式，详见 FFMPEG pixfmt.h 定义;</p> <p>void *data –系统内存地址，当为 null 时，表示该接口内部自己创建管理;</p> <p>long addr –设备内存地址;</p> <p>int fd –设备内存地址的句柄。如果为-1，表示设备内存由内部分配，反之则由 addr 参数给出。</p> <p>int* plane_stride –图像数据每层的每行 stride 数组;</p> <p>int* plane_size –图像数据每层的大小;</p> <p>int color_space –输入 image 的色彩空间，可以为 AVCOL_SPC_BT709 或 AVCOL_SPC_BT470，详见 FFMPEG pixfmt.h 定义，默认为 AVCOL_SPC_BT709;</p> <p>int color_range –输入 image 的色彩动态范围，可以为 AVCOL_RANGE_MPEG 或 AVCOL_RANGE_JPEG，详见 FFMPEG pixfmt.h 定义，默认为 AVCOL_RANGE_MPEG;</p> <p>int id –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0</p>
输出参数	无
返回值	AVFrame 结构体指针
说明	<p>1. 本接口支持创建以下图像格式的 AVFrame 数据结构: AV_PIX_FMT_GRAY8, AV_PIX_FMT_GBRP, AV_PIX_FMT_YUV420P, AV_PIX_FMT_NV12, AV_PIX_FMT_YUV422P horizontal, AV_PIX_FMT_YUV444P, AV_PIX_FMT_NV16</p> <p>2. 当设备内存和系统内存均有外部给出时，在 soc 模式下外部要保证两者地址的匹配，即系统内存是设备内存映射出来的虚拟地址；当设备内存由外部给出，系统内存为 null 时，该接口内部会自动创建系统内存；当设备内存没有给出，系统内存也为 null 时，本接口内部会自动创建；当设备内存没有给出，系统内存由外部给出时，本接口创建失败</p>

**AVFrame\* av::create(int height, int width, int id = 0)**

函数原型	AVFrame* av::create(int height, int width, int id = 0)
功能	AVFrame 的简易创建接口，所有内存均由内部创建管理，仅支持 YUV420P 格式
输入参数	int height –创建图像数据的高; int width –创建图像数据的宽; int id –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0
输出参数	无
返回值	AVFrame 结构体指针
说明	本接口仅支持创建 YUV420P 格式的 AVFrame 数据结构

**int av::copy(AVFrame \*src, AVFrame \*dst, int id)**

函数原型	int av::copy(AVFrame *src, AVFrame *dst, int id)
功能	AVFrame 的深度 copy 函数，将 src 的有效图像数据拷贝到 dst 中
输入参数	AVFrame *src –输入的 AVFrame 原始数据指针; int id –指定设备卡号，详见 5.1
输出参数	AVFrame *dst –输出的 AVFrame 目标数据指针
返回值	返回 copy 的有效图像数据个数，为 0 则没有发生拷贝
说明	1. 本接口仅支持同设备卡号内的图像数据拷贝，即 id 相同 2. 函数中的 id 仅需要指定设备卡号，不需要其他标志位

**int av::get\_scale\_and\_plane(int color\_format, int wscale[], int hscale[])**

函数原型	int av::get_scale_and_plane(int color_format, int wscale[], int hscale[])
功能	获取指定图像格式相对于 YUV444P 的宽高比例系数
输入参数	int color_format –指定图像格式，详见 FFMPEG pixfmt.h 定义
输出参数	int wscale[] –对应格式相对于 YUV444P 每一层的宽度比例; int hscale[] –对应格式相对于 YUV444P 每一层的高度比例
返回值	返回给定图像格式的 plane 层数
说明	

```
cv::Mat(int height, int width, int total, int _type, const size_t* _steps, void* _data, unsigned long
addr, int fd, SophonDevice device=SophonDevice())
```

函数原型	cv::Mat(int height, int width, int total, int _type, const size_t* _steps, void* _data, unsigned long addr, int fd, SophonDevice device=SophonDevice())
功能	新增的 Mat 构造接口。可以创建 opencv 标准格式或扩展的 YUV Mat 格式，并且系统内存和设备内存都允许通过外部分配给定
输入参数	int height –输入图像数据的高; int width –输入图像数据的宽; int total –内存大小，该内存可以为内部待分配的内存，或外部已分配内存的大小; int _type –Mat 类型，本接口只支持 CV_8UC1 或 CV_8UC3，扩展的 YUV Mat 的格式 _type 类型一律为 CV_8UC1; const size_t *_steps –所创建的图像数据的 step 信息，如果该指针为 null，则为 AUTO_STEP; void *_data –系统内存指针，如果为 null，则内部分配该内存; unsigned long addr –设备物理内存地址，任意值均被认为有效的物理地址; int fd –设备物理内存对应的句柄。如果为负，则设备物理内存存在内部分配管理; SophonDevice device –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0
输出参数	构造的标准 BGR 或扩展 YUV 的 Mat 数据类型
返回值	无
说明	1.SophonDevice 是为了避免 C++ 隐含类型匹配造成函数匹配失误而引入的类型，可以用 SophonDevice(int id) 直接从 5.1 节的 ID 转换过来 2. 当设备内存和系统内存均有外部给出时，在 soc 模式下外部要保证两者地址的匹配，即系统内存是设备内存映射出来的虚拟地址；当设备内存由外部给出，系统内存为 null 时，该接口内部会自动创建系统内存；当设备内存没有给出，系统内存也为 null 时，本接口内部会自动创建；当设备内存没有给出，系统内存由外部给出时，本接口创建的 Mat 在 soc 模式下只有系统内存

**Mat::Mat(SophonDevice device)**


函数原型	Mat::Mat(SophonDevice device)
功能	新增的 Mat 构造接口，指定该 Mat 的后续操作在给定的 device 设备上
输入参数	SophonDevice device - 指定设备卡号以及 HEAP 位置的标志，详见 5.1
输出参数	声明 Mat 数据类型
返回值	无
说明	<ol style="list-style-type: none"><li>1. 本构造函数仅初始化 Mat 内部的设备 index，并不实际创建内存</li><li>2. 本构造函数的最大作用是对于某些内部 create 内存的函数，可以通过这个构造函数，提前指定创建内存的设备号和 HEAP 位置，从而避免将大量的内存分配在默认的设备号 0 上</li></ol>



```
void Mat::create(int height, int width, int total, int _type, const size_t* _steps, void* _data, unsigned long addr, int fd, int id = 0)
```

函数原型	void Mat::create(int height, int width, int total, int type, const size_t* _steps, void* _data, unsigned long addr, int fd, int id = 0)
功能	Mat 分配内存接口，该接口系统内存和设备内存都允许通过外部分配给定，也可内部分配。
输入参数	int height –输入图像数据的高; int width –输入图像数据的宽; int total –内存大小，该内存可以为内部待分配的内存，或外部已分配内存的大小; int _type –Mat 类型，本接口只支持 CV_8UC1 或 CV_8UC3，扩展的 YUV Mat 的格式 _type 类型一律为 CV_8UC1; const size_t *_steps –所创建的图像数据的 step 信息，如果该指针为 null，则为 AUTO_STEP; void *_data –系统内存指针，如果为 null，则内部分配该内存; unsigned long addr –设备物理内存地址，任意值均被认为有效的物理地址; int fd –设备物理内存对应的句柄。如果为负，则设备物理内存存在内部分配管理; int id –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0
输出参数	无
返回值	无
说明	1. 扩展的内存分配接口，主要改进目的是允许外置指定设备物理内存，当设备或者系统内存由外部创建的时候，则外部必须负责该内存的释放，否则会造成内存泄漏 2. 当设备内存和系统内存均有外部给出时，在 soc 模式下外部要保证两者地址的匹配，即系统内存是设备内存映射出来的虚拟地址；当设备内存由外部给出，系统内存为 null 时，该接口内部会自动创建系统内存；当设备内存没有给出，系统内存也为 null 时，本接口内部会自动创建；当设备内存没有给出，系统内存由外部给出时，本接口创建的 Mat 在 soc 模式下只有系统内存

**void VideoWriter::write(InputArray image)**

函数原型	void VideoWriter::write(InputArray image)
功能	与 OpenCV 标准 VideoWriter::write 接口用法和功能一致 <b>在调用 write 将全部 frame 传入之后，需要额外调用一次 write，传入空的 Mat 进行 flush 操作</b>
输入参数	InputArray image –输入的图像数据 Mat 结构
返回值	无
使用示例	<pre>// General use, frame is the actual data you get from VideoCapture or  ↪somewhere write(frame); // Flush, empty is an empty data need not to be initialized Mat emptyMat; write(emptyMat); // 仅需要调用一次</pre>

**void VideoWriter::write(InputArray image, char \*data, int \*len, cv::CV\_RoiInfo \*roiinfo)**

函数原型	void VideoWriter::write(InputArray image, char *data, int len)
功能	新增的视频编码接口。与 OpenCV 标准 VideoWriter::write 接口不同，他提供了将编码视频数据输出到 buffer 的功能，便于后续处理 <b>在调用 write 将全部 frame 传入之后，需要额外调用多次 write，传入空的 Mat 进行 flush 操作，直到接收不到新数据 (len=0) 停止 flush</b>
输入参数	InputArray image –输入的图像数据 Mat 结构 cv::CV_RoiInfo *roiinfo(可选) - 输入的 roi 信息
输出参数	char *data –输出的编码数据缓存 int *len –输出的编码数据长度
返回值	无
使用示例	<pre>// General use, frame is the actual data you get from VideoCapture or  ↪somewhere write(frame, data, &amp;data_len); fwrite(data, 1, data_len, fp_out); //fp_out is the file to be written // Flush, empty is an empty data need not to be initialized Mat emptyMat; while(1){ //需要多次调用     writer.write(emptyMat, data, &amp;data_len);     if(len &gt; 0){         fwrite(data, 1, data_len, fp_out);         len = 0; //reset data_len     }else         break; //直到拿不出数据结束flush }</pre>

```
virtual bool VideoCapture::grab(char *buf, unsigned int len_in, unsigned int *len_out);
```

函数原型	bool VideoCapture::grab(char *buf, unsigned int len_in, unsigned int *len_out);
功能	新增的收流解码接口。与 OpenCV 标准 VideoWriter::grab 接口不同，他提供了将解码前的视频数据输出到 buf 的功能。
输入参数	char *buf –外部负责分配释放内存。 unsigned int len_in –buf 空间的大小。
输出参数	char *buf –输出解码前的视频数据。 int *len_out –输出的 buf 的实际大小。
返回值	true 表示收流解码成功； false 表示收流解码失败。

```
virtual bool VideoCapture::read_record(OutputArray image, char *buf, unsigned int len_in, unsigned int *len_out);
```

函数原型	bool VideoCapture::read_record(OutputArray image, char *buf, unsigned int len_in, unsigned int *len_out);
功能	新增的读取码流视频接口。他提供了将解码前的视频数据输出到 buf 的功能，将解码后的数据输出到 image。
输入参数	char *buf –外部负责分配释放内存。 unsigned int len_in –buf 空间的大小。
输出参数	OutputArray image –输出解码后的视频数据。 char *buf –输出解码前的视频数据。 int *len_out –输出的 buf 的实际大小。
返回值	true 表示收流解码成功； false 表示收流解码失败。

## 2.2.4 硬件 JPEG 解码器的 OpenCV 扩展

在 BM1686 系列芯片中，提供 JPEG 硬件编解码模块。为使用这些硬件模块，SDK 软件包中，扩展了 OpenCV 中与 JPEG 图片处理相关的 API 函数，如：cv::imread()、cv::imwrite()、cv::imdecode()、cv::imencode() 等。您在使用这些函数做 JPEG 编解码的时候，函数内部会自动调用底层的硬件加速资源，从而大幅度提高了编解码的效率。如果您想保持这些函数原始的 OpenCV API 使用习惯，可以略过本节介绍；但如果你还想了解一下我们提供的简单易用的扩展功能，这节可能对您非常有帮助。

## 输出 yuv 格式的图像数据

OpenCV 原生的 `cv::imread()`、`cv::imdecode()` API 函数执行 JPEG 图片的解码操作，返回一个 `Mat` 结构体，该 `Mat` 结构体中保存有 BGR packed 格式的图片数据，算能扩展的 API 函数功能可以返回 JPEG 图片解码后的原始的 YUV 格式数据。用法如下：

当这两个函数的第二个参数 `flags` 被设置成 `cv::IMREAD_AVFRAME` 时，表示解码后返回的 `Mat` 结构体 `out` 中保存着 YUV 格式的数据。具体是什么格式的 YUV 数据要根据 JPEG 文件的 `image` 格式而定。当 `flags` 被设置成其它值或者省略不设置时，表示解码输出 OpenCV 原生的 BGR packed 格式的 `Mat` 数据。解码器输入输出扩展数据格式说明如下表所示：

输入 Image 格式	输入 YUV 格式	FFMPEG 对应格式
I400	I400	AV_PIX_FMT_GRAY8
I420	NV12	AV_PIX_FMT_NV12
I422	NV16	AV_PIX_FMT_NV16
I444	I444 planar	AV_PIX_FMT_YUV444P

可以通过 `Mat::avFormat()` 扩展函数，得到当前数据所对应的具体的 FFmpeg 格式。可以通过 `Mat::avOK()` 扩展函数，得知 `cv::imdecode(buf, cv::IMREAD_AVFRAME, &out)` 解码返回的 `out`，是否是算能扩展的 `Mat` 数据格式。

另外在这两个接口中的 `flags` 增加 `cv::IMREAD_RETRY_SOFTDEC` 标志时会在硬件解码失败的情况下尝试切换软件解码，也可以通过设置环境变量 `OPENCV_RETRY_SOFTDEC=1` 实现此功能。

## 支持 YUV 格式的函数列表

目前算能 Opencv 已经支持 YUV Mat 扩展格式的函数接口列表如下：

- 视频解码类接口
  - VideoCapture 类的成员函数

这类成员函数如 `read`, `grab`，对于常用的 HEVC, H264 视频格式都使用了 BM1686 系列的硬件加速，并支持 YUV Mat 扩展格式。

- 视频编码类接口
  - VideoWriter 类的成员函数

这类成员函数如 `write`，对于常用的 HEVC, H264 视频格式已经使用了 BM1686 系列的硬件加速，并支持 YUV Mat 扩展格式。

- JPEG 编码类接口
- JPEG 解码类接口
  - `Imread`
  - `Imwrite`

- Imdecode
- Imencode

以上接口在处理 JPEG 格式的时候, 已经使用了 BM1686 系列的硬件加速功能, 并支持 YUV Mat 扩展格式。

- 图像处理类接口
  - cvtColor
  - resize

这两个接口在 BM1686 系列 SOC 模式下支持 YUV Mat 扩展格式, 并使用硬件加速进行了优化。

**尤其需要注意的是 cvtColor 接口, 只在 YUV 转换成 BGR 或者 GRAY 输出的时候支持硬件加速和 YUV Mat 的格式, 即只支持输入为 YUV Mat 格式, 并进行了硬件加速, 输出不支持 YUV Mat 格式。**

- line
- rectangle
- circle
- putText

以上四个接口均支持 YUV 扩展格式。注意, 这四个接口并没有采用硬件加速, 而是使用 CPU 对 YUV Mat 扩展格式进行的支持。

- 基本操作类接口
  - Mat 类部分接口
    - \* 创建释放接口: create, release, Mat 声明接口
    - \* 内存赋值接口: clone, copyTo, cloneAll, copyAllTo, assignTo, operator =,
    - \* 扩展 AV 接口: avOK, avComp, avRows, avCols, avFormat, avStep, avAddr

以上接口均支持 YUV 扩展格式, 尤其是 copyTo, clone 接口都采用硬件进行了加速。

- 扩展类接口
  - Bmcv 接口: 详见 opencv2/core/bmcv.hpp
  - AvFrame 接口: 详见 opencv2/core/av.hpp

以上算能扩展类接口, 均支持 YUV Mat 扩展格式, 并均针对硬件加速处理进行了优化。

**注意: 支持 YUV Mat 扩展格式的接口并不等价于使用了硬件加速, 部分接口是通过 CPU 处理来实现。这点需要特别注意。**

### 2.2.5 OpenCV 与 BMCV API 的调用原则

BMCV API 充分发挥了 BM1686 系列芯片中的硬件单元的加速能力，能提高数据处理的效率。而 OpenCV 软件提供了非常丰富的图像图形处理能力，将两者有机的结合起来，使客户开发既能利用 OpenCV 丰富的函数库，又能在硬件支持的功能上获得加速，是本节的主要目的。

在 BMCV API 和 OpenCV 函数以及数据类型的切换过程中，最关键是要尽量避免数据拷贝，使得切换代价最小。因此在调用流程中要遵循以下原则。

- 1) 由 OpenCV Mat 到 BMCV API 的切换，可以利用 toBMI() 函数，该函数以零拷贝的方式，将 Mat 中的数据转换成了 BMCV API 调用所需的 bm\_image 类型。
- 2) 当 BMCV API 需要切换到 OpenCV Mat 的时候，要将最后一步的操作通过 OpenCV 中的 bmcv 函数来实现。这样既完成所需的图像处理操作，同时也为后续 OpenCV 操作完成了数据类型准备。因为一般 OpenCV 都要求 BGR Pack 的色彩空间，所以一般用 toMat() 函数作为切换前的最后一步操作。
- 3) 一般神经网络处理的数据为不带 padding 的 RGB planar 数据，并且对于输入尺寸有特定的要求。因此建议将 resize() 函数作为调用神经网络 NPU 接口前的最后一步操作。
- 4) 当 crop、resize、color conversion 三个操作是连续的时候，强烈建议客户使用 convert() 函数，这可以在带宽优化和速度优化方面都获得理想的收益。即使后续可能还需要做一次拷贝，但因为拷贝发生在缩放之后的图像上，这种代价也是值得的。

### 2.2.6 OpenCV 中 GB28181 国标接口介绍

SOPHGO 复用 OpenCV 原生的 Cap 接口，通过对于 url 定义进行扩展，提供 GB28181 国标的播放支持。因此客户并不需要重新熟悉接口，只要对扩展的 url 定义进行理解，即可像播放 rtsp 视频一样，无缝的播放 GB28181 视频。

注意：国标中的 SIP 代理注册步骤，需要客户自己管理。当获取到前端设备列表后，可以直接用 url 的方式进行播放。

#### 国标 GB28181 支持的一般步骤

- 启动 SIP 代理（一般客户自己部署或者平台方提供）
  - 客户的下级应用平台注册到 SIP 代理
  - 客户应用获取前端设备列表，如下所示。其中，34010000001310000009 等为设备 20 位编码。
- ```
{ "devicelist" :
  [{ "id" : "34010000001310000009" }
   { "id" : "34010000001310000010" }
   { "id" : "34020000001310101202" } ] }
```
- 组织 GB28181 url 直接调用 OpenCV Cap 接口进行播放

## GB28181 url 格式定义

### UDP 实时流地址定义

```
gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid  
=34010000001310000009#localid=12478792871163624979#localip=172.  
10.18.201#localmediaport=20108:
```

```
gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid  
=34010000001310000009#localid=12478792871163624979#localip=172.  
10.18.201#localmediaport=20108:
```

#### 注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port  
deviceid:

前端设备 20 位编码

localid:

本地二十位编码, 可选项

localip:

本地 ip, 可选项

localmediaport:

媒体接收端的视频流端口, 需要做端口映射, 映射两个端口 (rtp:11801, rtcp:11802), 两个端口映射的 in 和 out 要相同。同一个核心板端口不可重复。

### UDP 回放流地址定义

```
gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid  
=35018284001310090010#devicetype=3#localid=12478792871163624979#localip=  
172.10.18.201#localmediaport=20108#begtime=20191018160000#endtime  
=20191026163713:
```

```
gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid  
=35018284001310090010#devicetype=3#localid=12478792871163624979#localip=  
172.10.18.201#localmediaport=20108#begtime=20191018160000#endtime  
=20191026163713:
```

注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port  
deviceid:

前端设备 20 位编码

devicetype:

录像存储类型

localid:

本地二十位编码, 可选项

localip:

本地 ip, 可选项

localmediaport:

媒体接收端的视频流端口, 需要做端口映射, 映射两个端口 (rtp:11801, rtcp:11802), 两个端口映射的 in 和 out 要相同。同一个核心板端口不可重复。

begtime:

录像起始时间

endtime:

录像结束时间

## TCP 实时流地址定义

gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid  
=35018284001310090010#localid=12478792871163624979#localip=172.10.18.201:

gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid  
=35018284001310090010#localid=12478792871163624979#localip=172.10.18.201:

注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port  
deviceid:

前端设备 20 位编码

localid:



本地二十位编码，可选项

localip:

本地 ip，可选项

### TCP 回放流地址定义

```
gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid=35018284001310090010#devicetype=3#localid=12478792871163624979#localip=172.10.18.201#begtime=20191018160000#endtime=20191026163713:
```

```
gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid=35018284001310090010#devicetype=3#localid=12478792871163624979#localip=172.10.18.201#begtime=20191018160000#endtime=20191026163713:
```

注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port  
deviceid:

前端设备 20 位编码

devicetype:

录像存储类型

localid:

本地二十位编码，可选项

localip:

本地 ip，可选项

begtime:

录像起始时间

endtime:

录像结束时间

## 2.3 SOPHGO FFMPEG 使用指南

### 2.3.1 前言

BM1686 系列芯片中，有一个 8 核的 A53 处理器，同时还内置有视频、图像相关硬件加速模块。在 SOPHGO 提供的 FFMPEG SDK 开发包中，提供了对这些硬件模块的接口。其中，通过这些硬件接口，提供了如下模块：硬件视频解码器、硬件视频编码器、硬件 JPEG 解码器、硬件 JPEG 编码器、硬件 scale filter、hwupload filter、hwdownload filter。

FFMPEG SDK 开发包符合 FFMPEG hwaccel 编写规范，实现了视频转码硬件加速框架，实现了硬件内存管理、各个硬件处理模块流程的组织等功能。同时 FFMPEG SDK 也提供了与通常 CPU 解码器兼容的接口，以匹配部分客户的使用习惯。这两套接口我们称之为 HWAaccel 接口和常规接口，他们底层共享 BM1686 硬件加速模块，在性能上是相同的。区别仅在于 1) HWAaccel 需要初始化硬件设备 2) HWAaccel 接口只面向设备内存，而常规接口同时分配了设备内存和系统内存 3) 他们的参数配置和接口调用上有轻微差别。

下面描述中，如非特殊说明，对常规接口和 HWAaccel 接口都适用。

### 2.3.2 硬件视频解码器

BM1686 系列支持 H.264 和 H.265 硬件解码。硬件解码器性能详情如下表所述。

| Standard   | Profile      | Level | Max Resolution | Min Resolution | Bit rate |
|------------|--------------|-------|----------------|----------------|----------|
| H.264/AVC  | BP/CBP/MP/HP | 4.1   | 8192x4096      | 16x16          | 50Mbps   |
| H.265/HEVC | Main/Main10  | L5.1  | 8192x4096      | 16x16          | N/A      |

在 SophGo 的 FFMPEG 发布包中，H.264 硬件视频解码器的名字为 h264\_bm，H.265 硬件视频解码器的名字为 hevc\_bm。可通过如下命令，来查询 FFMPEG 支持的编码器。

```
$ ffmpeg -decoders | grep _bm
```

#### 硬件视频解码器支持的选项

FFMPEG 中，BM1686 系列的硬件解码器提供了一些额外选项，可以通过如下命令查询。

```
$ ffmpeg -h decoder=h264_bm
```

```
$ ffmpeg -h decoder=hevc_bm
```

这些选项可以使用 av\_dict\_set API 来设置。在设置之前，需要对对这些选项有正确的理解。下面详细解释一下这些选项。

output\_format:

- 输出数据的格式。
- 设为 0，则输出线性排列的未压缩数据；设为 101，则输出压缩数据。

- 缺省值为 0。
- 推荐设置为 101，输出压缩数据。可以节省内存、节省带宽。输出的压缩数据，可以调用后面介绍的 `scale_bm` filter 解压缩成正常的 YUV 数据。具体可参考应用示例中的示例 1。

`cbcr_interleave`:

- 硬件视频解码器解码输出的帧色度数据是否是交织格式。
- 设为 1，则输出为 semi-planar yuv 图像，譬如 nv12；设为 0，则输出 planar yuv 图像，譬如 yuv420p。
- 缺省值为 1。

`extra_frame_buffer_num`:

- 硬件视频解码器额外提供硬件帧缓存数量。
- 缺省值为 2。最小值为 1。

`skip_non_idr`:

- 跳帧模式。0，关闭；1，跳过 Non-RAP 帧；2，跳过非参考帧。
- 缺省值为 0。

`handle_packet_loss`

- 出错时，对 H.264, H.265 解码器使能丢包处理。0，不做丢包处理；1，进行丢包处理。
- 缺省值为 0。

`zero_copy`:

- 将设备上的帧数据直接拷贝到 AVFrame 的 `data[0]-data[3]` 所自动申请的系统内存里。1，关闭拷贝；0，使能拷贝。
- 缺省值为 1。

### 2.3.3 硬件视频编码器

从 BM1686 开始首次添加了硬件视频编码器。支持 H.264/AVC 和 H.265/HEVC 视频编码。

BM1686 硬件编码器设计的能力为：能够实时编码一路 1080P30 的视频。具体指标如下：

#### H.265 编码器:

- Capable of encoding HEVC Main/Main10/MSP(Main Still Picture) Profile @ L5.1 High-tier

#### H.264 编码器:

- Capable of encoding Baseline/Constrained Baseline/Main/High/High 10 Profiles Level @ L5.2

#### 通用指标

- 最大分辨率: 8192x8192
- 最小分辨率: 256x128
- 编码图像宽度须为 8 的倍数
- 编码图像高度宽度须为 8 的倍数

在 SophGo 的 FFMPEG 发布包中, H.264 硬件视频编码器的名字为 h264\_bm, H.265 硬件视频编码器的名字为 h265\_bm 或 hevc\_bm。可通过如下命令, 来查询 FFMPEG 支持的编码器。

```
$ ffmpeg -encoders
```

### 硬件视频编码器支持的选项

FFMPEG 中, 硬件视频编码器提供了一些额外选项, 可以通过如下命令查询。

```
$ ffmpeg -h encoder=h264_bm
```

```
$ ffmpeg -h encoder=hevc_bm
```

BM1686 硬件视频编码器支持如下选项:

preset: 预设编码模式。推荐通过 enc-params 设置。

- 0 - fast, 1 - medium, 2 - slow。
- 缺省值为 2。

gop\_preset: gop 预设索引值。推荐通过 enc-params 设置。

- 1: all I, gopsize 1
- 2: IPP, cyclic gopsize 1
- 3: IBB, cyclic gopsize 1
- 4: IBPBP, cyclic gopsize 2
- 5: IBBBP, cyclic gopsize 4
- 6: IPPPP, cyclic gopsize 4
- 7: IB BBB, cyclic gopsize 4
- 8: random access, IB BBB BBBB, cyclic gopsize 8

qp:

- 恒量化参数的码率控制方法
- 取值范围为 0 至 51

perf:

- 用于指示是否需要测试编码器性能
- 取值范围为 0 或 1。

enc-params:

- 用于设置视频编码器内部参数。
- 支持的编码参数: preset, gop\_preset, qp, bitrate, mb\_rc, delta\_qp, min\_qp, max\_qp, bg, nr, deblock, weightp
- 编码参数 preset: 取值范围为 fast, medium, slow 或者是 0, 1, 2
- 编码参数 gop\_preset: gop 预设索引值。参考上面已有详细解释。
  - 1: all I, gopsize 1
  - 2: IPP, cyclic gopsize 1
  - 3: IBB, cyclic gopsize 1
  - 4: IBPBP, cyclic gopsize 2
  - 5: IBBBP, cyclic gopsize 4
  - 6: IPPPP, cyclic gopsize 4
  - 7: IB BBB, cyclic gopsize 4
  - 8: random access, IB BBB BBBB, cyclic gopsize 8
- 编码参数 qp: 恒定量化参数, 取值范围为 [0, 51]。当该值有效时, 关闭码率控制算法, 用固定的量化参数编码。
- 编码参数 bitrate: 用于编码所指定的码率。单位是 Kbps, 1Kbps=1000bps。当指定改参数时, 请不要设置编码参数 qp。
- 编码参数 mb\_rc: 取值范围 0 或 1。当设为 1 时, 开启宏块级码率控制算法; 当设为 0 时, 开启帧级码率控制算法。
- 编码参数 delta\_qp: 用于码率控制算法的 QP 最大差值。该值太大影响视频主观质量。太小影响码率调整的速度。
- 编码参数 min\_qp 和 max\_qp: 码率控制算法中用于控制码率和视频质量的最小量化参数和最大量化参数。取值范围 [0, 51]。
- 编码参数 bg: 是否开启背景检测。取值范围 0 或 1。
- 编码参数 nr: 是否开启降噪算法。取值范围 0 或 1。
- 编码参数 deblock: 是否开启环状滤波器。有如下几种用法:
  - 关闭环状滤波器 “deblock=0” 或 “no-deblock”。
  - 简单开启环状滤波器, 使用缺省环状滤波器参数” deblock=1”。
  - 开启环状滤波器并设置参数, 譬如” deblock=6,6”。
- 编码参数 weightp: 是否开启 P 帧、B 帧加权预测。取值范围 0 或 1。

is\_dma\_buffer:

- 用于提示编码器, 输入的帧缓存是否为设备上的连续物理内存地址。

- 在 SoC 模式, 值 0 表示输入的是设备内存的虚拟地址。值 1 表示, 输入的是设备上的连续物理地址。
- 缺省值为 1。
- 仅适用于常规接口。

### 2.3.4 硬件 JPEG 解码器

在 BM1686 系列芯片中, 硬件 JPEG 解码器提供硬件 JPEG 图像解码输入能力。这里介绍一下, 如何通过 FFMPEG 来实现硬件 JPEG 解码。

在 FFMPEG 中, 硬件 JPEG 解码器的名称为 jpeg\_bm。可以通过如下命令, 来查看 FFMPEG 中是否有 jpeg\_bm 解码器。

```
$ ffmpeg -decoders | grep jpeg_bm
```

#### 硬件 JPEG 解码器支持的选项

FFMPEG 中, 可以通过如下命令, 来查看 jpeg\_bm 解码器支持的选项

```
$ ffmpeg -h decoder=jpeg_bm
```

解码选项的说明如下。硬件 JPEG 解码器中这些选项, 可以使用 `av_dict_set()` API 函数对其进行重置。

`bs_buffer_size`: 用于设置硬件 JPEG 解码器中输入比特流的缓存大小 (KBytes)。

- 取值范围 (0 到 INT\_MAX)
- 缺省值 5120

`cbr_interleave`: 用于指示 JPEG 解码器输出的帧数据中色度数据是否为交织的格式。

- 0: 输出的帧数据中色度数据为 planar 的格式
- 1: 输出的帧数据中色度数据为 interleave 的格式
- 缺省值为 0

`num_extra_framebuffers`: JPEG 解码器需要的额外帧缓存数量

- 对于 Still JPEG 的输入, 建议该值设为 0
- 对于 Motion JPEG 的输入, 建议该值至少为 2
- 取值范围 (0 到 INT\_MAX)
- 缺省值为 2

### 2.3.5 硬件 JPEG 编码器

在 BM1686 系列芯片中，硬件 JPEG 编码器提供硬件 JPEG 图像编码输出能力。这里介绍一下，如何通过 FFMPEG 来实现硬件 JPEG 编码。

在 FFMPEG 中，硬件 JPEG 编码器的名称为 jpeg\_bm。可以通过如下命令，来查看 FFMPEG 中是否有 jpeg\_bm 编码器。

```
$ ffmpeg -encoders | grep jpeg_bm
```

#### 硬件 JPEG 编码器支持的选项

FFMPEG 中，可以通过如下命令，来查看 jpeg\_bm 编码器支持的选项

```
$ ffmpeg -h encoder=jpeg_bm
```

编码选项的说明如下。硬件 JPEG 编码器中这些选项，可以使用 av\_dict\_set() API 函数对其进行重置。

is\_dma\_buffer:

- 用于提示编码器，输入的帧缓存是否为设备上的连续物理内存地址。
- 在 SoC 模式，值 0 表示输入的是设备内存的虚拟地址。值 1 表示，输入的是设备上的连续物理地址。
- 缺省值为 1。
- 仅适用于常规接口。

### 2.3.6 硬件 scale filter

BM1686 系列硬件 scale filter 用于将输入的图像进行”缩放/裁剪/补边”操作。譬如，转码应用。在将 1080p 的视频解码后，使用硬件 scale 缩放成 720p 的，再进行压缩输出。

| 内容   | 最大分辨率       | 最小分辨率 | 放大倍数 |
|------|-------------|-------|------|
| 硬件限制 | 4096 * 4096 | 8*8   | 32   |

在 FFMPEG 中，硬件 scale filter 的名称为 scale\_bm。

```
$ ffmpeg -filters | grep bm
```

**硬件 scale filter 支持的选项**

FFMPEG 中, 可以通过如下命令, 来查看 scaler\_bm 编码器支持的选项

```
$ ffmpeg -h filter=scale_bm
```

scale\_bm 选项的说明如下:

w:

- 缩放输出视频的宽度。请参考 ffmpeg scale filter 的用法。

h:

- 缩放输出视频的高度。请参考 ffmpeg scale filter 的用法。

format:

- 缩放输出视频的像素格式。请参考 ffmpeg scale filter 的用法。
- 输入输出支持的格式详见附表 7.1。
- 缺省值” none”。即输出像素格式为系统自动。输入为 yuv420p, 输出为 yuv420p; 输入为 yuvj420p, 输出为 yuvj420p。输入为 nv12 时, 缺省输出为 yuv420p。
- 在 HWAccel 框架下: 支持 nv12 到 yuv420p、nv12 到 yuvj420p、yuv420p 到 yuvj420p、yuvj422p 到 yuvj420p、yuvj422p 到 yuv420p 的格式转换。在不启用 HWAccel 框架的正常模式下支持情况见附表 7.1。

| 输入         | 输出      | 是否持缩放 | 是否支持颜色转换 |
|------------|---------|-------|----------|
| GRAY8      | GRAY8   | 是     | 是        |
| NV12 (压缩)  | YUV420P | 是     | 是        |
|            | YUV422P | 否     | 是        |
|            | YUV444P | 是     | 是        |
|            | BGR     | 是     | 是        |
|            | RGB     | 是     | 是        |
|            | RGBP    | 是     | 是        |
| NV12 (非压缩) | BGRP    | 是     | 是        |
|            | YUV420P | 是     | 是        |
|            | YUV422P | 否     | 是        |
|            | YUV444P | 是     | 是        |
|            | BGR     | 是     | 是        |
|            | RGB     | 是     | 是        |
| YUV420P    | RGBP    | 是     | 是        |
|            | BGRP    | 是     | 是        |
|            | YUV420P | 是     | 是        |
|            | YUV422P | 否     | 是        |
|            | YUV444P | 是     | 是        |
|            | BGR     | 是     | 是        |
|            | RGB     | 是     | 是        |

续下页



表 2.1 – 接上页

| 输入        | 输出      | 是否持缩放 | 是否支持颜色转换 |
|-----------|---------|-------|----------|
| YUV422P   | RGBP    | 是     | 是        |
|           | BGRP    | 是     | 是        |
|           | YUV420P | 是     | 是        |
|           | YUV422P | 否     | 否        |
|           | YUV444P | 否     | 否        |
|           | BGR     | 是     | 是        |
| YUV444P   | RGB     | 是     | 是        |
|           | RGBP    | 是     | 是        |
|           | BGRP    | 是     | 是        |
|           | YUV420P | 是     | 是        |
|           | YUV422P | 否     | 是        |
|           | YUV444P | 是     | 是        |
| BGR、RGB   | BGR     | 是     | 是        |
|           | RGB     | 是     | 是        |
|           | RGBP    | 是     | 是        |
|           | BGRP    | 是     | 是        |
|           | YUV420P | 是     | 是        |
|           | YUV422P | 否     | 是        |
| RGBP、BGRP | YUV444P | 是     | 是        |
|           | BGR     | 是     | 是        |
|           | RGB     | 是     | 是        |
|           | RGBP    | 是     | 是        |
|           | BGRP    | 是     | 是        |
|           | YUV420P | 是     | 是        |
|           | YUV422P | 否     | 是        |
|           | YUV444P | 是     | 是        |
|           | BGR     | 是     | 是        |
|           | RGB     | 是     | 是        |
|           | RGBP    | 是     | 是        |
|           | BGRP    | 是     | 是        |

表 7.1 scale\_bm 像素格式支持列表

opt:

- 缩放操作 (from 0 to 2) (default 0)
- 值 0 - 仅支持缩放操作。缺省值。
- 值 1 - 支持缩放 + 自动裁剪操作。命令行参数中可用 crop 来表示。
- 值 2 - 支持缩放 + 自动补黑边操作。命令行参数中可用 pad 来表示。

flags:

- 缩放方法 (from 0 to 2) (default 1)
- 值 0 - nearest 滤波器。命令行参数中, 可用 nearest 来表示。

- 值 1 - bilinear 滤波器。命令行参数中，可用 bilinear 来表示。
- 值 2 - bicubic 滤波器。命令行参数中，可用 bicubic 来表示。

sophon\_idx:

- 设备 ID，从 0 开始。

zero\_copy:

- 值 0 - 表示 scale\_bm 的输出 AVFrame 将同时包含设备内存和主机内存指针，兼容性最好，性能稍有下降。缺省为 0
- 值 1 - 表示 scale\_bm 的输出到下一级的 AVFrame 中将只包含有效设备地址，不会对数据进行从设备内存到系统内存的同步。建议对于下一级接使用 SOPHGO 的编码/filter 的情况，可以选择设置为 1，其他建议设置为 0。
- 缺省为 0

### 2.3.7 AVFrame 特殊定义说明

遵从 FFMPEG 的规范，硬件解码器是通过 AVFrame 来提供输出的，硬件编码器是通过 AVFrame 来提供输入的。因此，在通过 API 方式，调用 FFMPEG SDK、进行硬件编解码处理时，需要注意到 AVFrame 的如下特殊规定。AVFrame 是线性 YUV 输出。在 AVFrame 中，data 为数据指针，用于保存物理地址，linesize 为每个平面的线跨度。

#### 硬件解码器输出的 avframe 接口定义

##### 常规接口

data 数组的定义

| 下标 | 说明                                                              |
|----|-----------------------------------------------------------------|
| 0  | Y 的虚拟地址                                                         |
| 1  | cbcr_interleave=1 时 CbCr 的虚拟地址;<br>cbcr_interleave=0 时 Cb 的虚拟地址 |
| 2  | cbcr_interleave=0 时 Cr 的虚拟地址                                    |
| 3  | 未使用                                                             |
| 4  | Y 的物理地址                                                         |
| 5  | cbcr_interleave=1 时 CbCr 的物理地址;<br>cbcr_interleave=0 时 Cb 的物理地址 |
| 6  | cbcr_interleave=0 时 Cr 的物理地址                                    |
| 7  | 未使用                                                             |

linesize 数组的定义

| 下标 | 说明                                                                  |
|----|---------------------------------------------------------------------|
| 0  | Y 的虚拟地址的跨度                                                          |
| 1  | cbr_interleave=1 时 CbCr 的虚拟地址的跨度;<br>cbr_interleave=0 时 Cb 的虚拟地址的跨度 |
| 2  | cbr_interleave=0 时 Cr 的虚拟地址的跨度                                      |
| 3  | 未使用                                                                 |
| 4  | Y 的物理地址的跨度                                                          |
| 5  | cbr_interleave=1 时 CbCr 的物理地址的跨度;<br>cbr_interleave=0 时 Cb 的物理地址的跨度 |
| 6  | cbr_interleave=0 时 Cr 的物理地址的跨度                                      |
| 7  | 未使用                                                                 |

## HWAccel 接口

data 数组的定义

| 下标 | 未压缩格式说明                                                       | 压缩格式明          |
|----|---------------------------------------------------------------|----------------|
| 0  | Y 的物理地址                                                       | 压缩的亮度数据的物理地址   |
| 1  | cbr_interleave=1 时 CbCr 的物理地址;<br>cbr_interleave=0 时 Cb 的物理地址 | 压缩的色度数据的物理地址   |
| 2  | cbr_interleave=0 时 Cr 的物理地址                                   | 亮度数据的偏移量表的物理地址 |
| 3  | 保留                                                            | 色度数据的偏移量表的物理地址 |
| 4  | 保留                                                            | 保留             |

linesize 数组的定义

| 下标 | 未压缩格式说明                                                             | 压缩格式说明    |
|----|---------------------------------------------------------------------|-----------|
| 0  | Y 的物理地址的跨度                                                          | 亮度数据的跨度   |
| 1  | cbr_interleave=1 时 CbCr 的物理地址的跨度;<br>cbr_interleave=0 时 Cb 的物理地址的跨度 | 色度数据的跨度   |
| 2  | cbr_interleave=0 时 Cr 的物理地址的跨度                                      | 亮度偏移量表的大小 |
| 3  | 未使用色                                                                | 偏移量表的大小   |

**硬件编解码器输入的 avframe 接口定义****常规接口**

data 数组的定义

| 下标 | 说明       |
|----|----------|
| 0  | Y 的虚拟地址  |
| 1  | Cb 的虚拟地址 |
| 2  | Cr 的虚拟地址 |
| 3  | 保留       |
| 4  | Y 的物理地址  |
| 5  | Cb 的物理地址 |
| 6  | Cr 的物理地址 |
| 7  | 未使用      |

linesize 数组的定义

| 下标 | 说明          |
|----|-------------|
| 0  | Y 的虚拟地址的跨度  |
| 1  | Cb 的虚拟地址的跨度 |
| 2  | Cr 的虚拟地址的跨度 |
| 3  | 未使用         |
| 4  | Y 的物理地址的跨度  |
| 5  | Cb 的物理地址的跨度 |
| 6  | Cr 的物理地址的跨度 |
| 7  | 未使用         |

**HWAccel 接口**

data 数组的定义

| 下标 | 说明       |
|----|----------|
| 0  | Y 的物理地址  |
| 1  | Cb 的物理地址 |
| 2  | Cr 的物理地址 |
| 3  | 保留       |
| 4  | 保留       |

linesize 数组的定义

| 下标 | 说明          |
|----|-------------|
| 0  | Y 的物理地址的跨度  |
| 1  | Cb 的物理地址的跨度 |
| 2  | Cr 的物理地址的跨度 |
| 3  | 未使用         |

### 硬件 filter 输入输出的 AVFrame 接口定义

1. 在不启用 HWAaccel 加速功能时, AVFrame 接口定义采用常规接口的内存布局。

data 数组的定义

| 下标 | 说明       |
|----|----------|
| 0  | Y 的虚拟地址  |
| 1  | Cb 的虚拟地址 |
| 2  | Cr 的虚拟地址 |
| 3  | 保留       |
| 4  | Y 的物理地址  |
| 5  | Cb 的物理地址 |
| 6  | Cr 的物理地址 |
| 7  | 未使用      |

linesize 数组的定义

| 下标 | 说明          |
|----|-------------|
| 0  | Y 的虚拟地址的跨度  |
| 1  | Cb 的虚拟地址的跨度 |
| 2  | Cr 的虚拟地址的跨度 |
| 3  | 未使用         |
| 4  | Y 的物理地址的跨度  |
| 5  | Cb 的物理地址的跨度 |
| 6  | Cr 的物理地址的跨度 |
| 7  | 未使用         |

### 2.HWAaccel 接口下 AVFrame 接口定义

data 数组的定义

| 下标 | 说明      | 压缩格式的输入接口      |
|----|---------|----------------|
| 0  | Y 的物理地址 | 压缩的亮度数据的物理地址   |
| 1  | Cb 物理地址 | 压缩的色度数据的物理地址   |
| 2  | Cr 物理地址 | 亮度数据的偏移量表的物理地址 |
| 3  | 保留      | 色度数据的偏移量表的物理地址 |
| 4  | 保留      | 保留             |

linesize 数组的定义

| 下标 | 说明         | 缩格式的输入接口  |
|----|------------|-----------|
| 0  | Y 物理地址的跨度  | 亮度数据的跨度   |
| 1  | Cb 物理地址的跨度 | 色度数据的跨度   |
| 2  | Cr 物理地址的跨度 | 亮度偏移量表的大小 |
| 3  | 未使用        | 色度偏移量表的大小 |

### 2.3.8 硬件加速在 FFMPEG 命令中的应用示例

下面同时给出常规模式和 HWAcel 模式对应的 FFMPEG 命令行参数。

为便于理解，这里汇总说明：

- 常规模式下，bm 解码器的输出内存是否同步到系统内存上，用 zero\_copy 控制，默认为 1。
- 常规模式下，bm 编码器的输入内存存在系统内容还是设备内存上，用 is\_dma\_buffer 控制，默认值为 1。
- 常规模式下，bm 滤波器会自动判断输入内存的同步，输出内存是否同步到系统内存，用 zero\_copy 控制，默认值为 0。
- HWAcel 模式下，设备内存和系统内存的同步用 hwupload 和 hwdownload 来控制。
- 常规模式下，用 sophon\_idx 来指定设备，默认为 0；HWAcel 模式下用 hwaccel\_device 来指定。

**示例 1**

使用设备 0。解码 H.265 视频，输出 compressed frame buffer，scale\_bm 解压缩 compressed frame buffer 并缩放成 CIF，然后编码成 H.264 码流。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:zero_copy=1" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale.264
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale.264
```

**示例 2**

使用设备 0。解码 H.265 视频，按比例缩放并自动裁剪成 CIF，然后编码成 H.264 码流。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=crop:zero_copy=1" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_crop.264
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=crop" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_crop.264
```

### 示例 3

使用设备 0。解码 H.265 视频，按比例缩放并自动补黑边成 CIF，然后编码成 H.264 码流。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \  
-vf "scale_bm=352:288:opt=pad:zero_copy=1" \  
-c:v h264_bm -g 256 -b:v 256K \  
-y wkc_100_cif_scale_pad.264
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \  
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \  
-vf "scale_bm=352:288:opt=pad" \  
-c:v h264_bm -g 256 -b:v 256K \  
-y wkc_100_cif_scale_pad.264
```

### 示例 4

演示视频截图功能。使用设备 0。解码 H.265 视频，按比例缩放并自动补黑边成 CIF，然后编码成 jpeg 图片。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \  
-vf "scale_bm=352:288:opt=pad:format=yuvj420p:zero_copy=1" \  
-c:v jpeg_bm -vframes 1 \  
-y wkc_100_cif_scale.jpeg
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \  
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \  
-vf "scale_bm=352:288:opt=pad:format=yuvj420p" \  
-c:v jpeg_bm -vframes 1 \  
-y wkc_100_cif_scale.jpeg
```



### 示例 5

演示视频转码 + 视频截图功能。使用设备 0。硬件解码 H.265 视频，缩放成 CIF，然后编码成 H.264 码流；同时缩放成 720p，然后编码成 JPEG 图片。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-filter_complex "[0:v]scale_bm=352:288:zero_copy=1[img1];[0:v]scale_bm=1280:720:format=
\
yuvj420p:zero_copy=1[img2]" -map '[img1]' -c:v h264_bm -b:v 256K -y img1.264 \
-map '[img2]' -c:v jpeg_bm -vframes 1 -y img2.jpeg
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-filter_complex "[0:v]scale_bm=352:288[img1];[0:v]scale_bm=1280:720:format=yuvj420p[img2]"
\
-map '[img1]' -c:v h264_bm -b:v 256K -y img1.264 \
-map '[img2]' -c:v jpeg_bm -vframes 1 -y img2.jpeg
```

### 示例 6

演示 hwdownload 功能。硬件解码 H.265 视频，然后下载存储成 YUV 文件。

Filter hwdownload 专门为 HWAccel 接口服务，用于设备内存和系统内存的同步。在常规模式中，这步可以通过 codec 中指定 zero\_copy 选项来实现，因此不需要 hwdownload 滤波器。

常规模式：

```
$ ffmpeg -c:v hevc_bm -cbcr_interleave 0 -zero_copy 0 \
-i src/wkc_100.265 -y test_transfer.yuv
```

HWAccel 模式

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -cbcr_interleave 0 -i src/wkc_100.265 \
-vf "hwdownload,format=yuv420p bmcodec" \
-y test_transfer.yuv
```

### 示例 7

演示 hwdownload 功能。硬件解码 H.265 视频，缩放成 CIF 格式，然后下载存储成 YUV 文件。

在常规模式中，scale\_bm 会自动根据 filter 的链条判定是否同步内存，因此不需要 hwdownload。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \  
-vf "scale_bm=352:288,format=yuv420p" \  
-y test_transfer_cif.yuv
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \  
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \  
-vf "scale_bm=352:288,hwdownload,format=yuv420p bmcodec" \  
-y test_transfer_cif.yuv
```

### 示例 8

演示 hwupload 功能。使用设备 0。上传 YUV 视频，然后编码 H.264 视频。

Filter hwupload 专门为 HWAccel 接口服务，用于设备内存和系统内存的同步。在常规模式中，这步可以通过编码器中指定 is\_dma\_buffer 选项来实现，因此不需要 hwupload 滤波器。

常规模式：

```
$ ffmpeg -s 1920x1080 -pix_fmt yuv420p -i test_transfer.yuv \  
-c:v h264_bm -b:v 3M -is_dma_buffer 0 -y test_transfer.264
```

HWAccel 模式：

```
$ ffmpeg -init_hw_device bmcodec=foo:0 \  
-s 1920x1080 -i test_transfer.yuv \  
-filter_hw_device foo -vf "format=yuv420p bmcodec,hwupload" \  
-c:v h264_bm -b:v 3M -y test_transfer.264
```

这里 foo 为设备 0 的别名。

### 示例 9

演示 hwupload 功能。使用设备 1。上传 YUV 视频，并缩放成 CIF，然后编码 H.264 视频。

常规模式：

```
$ ffmpeg -s 1920x1080 -i test_transfer.yuv \  
-vf "scale_bm=352:288:sophon_idx=1:zero_copy=1" \  
-c:v h264_bm -b:v 256K -sophon_idx 1 \  
-y test_transfer_cif.264
```

说明：1) 这里不指定-pix\_fmt yuv420p 是因为默认输入为 yuv420p 格式

2) 常规模式下,bm\_scale filter, decoder, encoder 通过参数 sophon\_idx 来指定使用哪个设备  
HWAaccel 模式：

```
$ ffmpeg -init_hw_device bmcodec=foo:1 \  
-s 1920x1080 -i test_transfer.yuv \  
-filter_hw_device foo \  
-vf "format=yuv420p bmcodec,hwupload,scale_bm=352:288" \  
-c:v h264_bm -b:v 256K -y test_transfer_cif.264
```

说明：这里 foo 为设备 1 的别名，HWAaccel 模式下通过 init\_hw\_device 来指定使用具体的硬件设备。

### 示例 10

演示 hwdownload 功能。硬件解码 YUVJ444P 的 JPEG 视频，然后下载存储成 YUV 文件。

常规模式：

```
$ ffmpeg -c:v jpeg_bm -zero_copy 0 -i src/car/1920x1080_yuvj444.jpg \  
-y car_1080p_yuvj444_dec.yuv
```

HWAaccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \  
-c:v jpeg_bm -i src/car/1920x1080_yuvj444.jpg \  
-vf "hwdownload,format=yuvj444p bmcodec" \  
-y car_1080p_yuvj444_dec.yuv
```

**示例 11**

演示 hwupload 功能。使用设备 1。上传 YUVJ444P 图像数据，然后编码 JPEG 图片。

常规模式：

```
$ ffmpeg -s 1920x1080 -pix_fmt yuvj444p -i car_1080p_yuvj444.yuv \
-c:v jpeg_b -sophon_idx 1 -is_dma_buffer 0 \
-y car_1080p_yuvj444_enc.jpg
```

HWAccel 模式：

```
$ ffmpeg -init_hw_device bmcodec=foo:1 \
-s 1920x1080 -pix_fmt yuvj444p -i car_1080p_yuvj444.yuv \
-filter_hw_device foo -vf 'format=yuvj444p bmcodec,hwupload' \
-c:v jpeg_b -y car_1080p_yuvj444_enc.jpg
```

这里 foo 为设备 1 的别名。

**示例 12**

演示软解码和硬编码混合使用的方法。使用设备 2。使用 ffmpeg 自带的 h264 软解码器，解码 H.264 视频，上传解码后数据到芯片 2，然后编码 H.265 视频。

常规模式：

```
$ ffmpeg -c:v h264 -i src/1920_18MG.mp4 \
-c:v h265_b -is_dma_buffer 0 -sophon_idx 2 -g 256 -b:v 5M \
-y test265.mp4
```

HWAccel 模式：

```
$ ffmpeg -init_hw_device bmcodec=foo:2 \
-c:v h264 -i src/1920_18MG.mp4 \
-filter_hw_device foo -vf 'format=yuv420p bmcodec,hwupload' \
-c:v h265_b -g 256 -b:v 5M \
-y test265.mp4
```

这里 foo 为设备 2 的别名。

**示例 13**

演示使用 `enc-params` 设置视频编码器的方法。使用设备 0。解码 H.265 视频，缩放成 CIF，然后编码成 H.264 码流。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:zero_copy=1" \
-c:v h264_bm -g 50 -b:v 32K \
-enc-params "gop_preset=2:mb_rc=1:delta_qp=3:min_qp=20:max_qp=40" \
-y wkc_100_cif_scale_ipp_32Kbps.264
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288" \
-c:v h264_bm -g 50 -b:v 32K \
-enc-params "gop_preset=2:mb_rc=1:delta_qp=3:min_qp=20:max_qp=40" \
-y wkc_100_cif_scale_ipp_32Kbps.264
```

**示例 14**

使用设备 0。解码 H.265 视频，使用 `bilinear` 滤波器，按比例缩放成 CIF，并自动补黑边，然后编码成 H.264 码流。

常规模式：

```
$ ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=pad:flags=bilinear:zero_copy=1" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_pad.264
```

HWAccel 模式：

```
$ ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=pad:flags=bilinear" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_pad.264
```

### 2.3.9 通过调用 API 方式来使用硬件加速功能

examples/multimedia/ff\_bmcv\_transcode/例子演示了使用 ffmpeg 做编解码，用 bmcv 做图像处理的整个流程。

examples/multimedia/ff\_video\_decode/例子演示了使用 ffmpeg 做解码的流程。

examples/multimedia/ff\_video\_encode/例子演示了使用 ffmpeg 做编码的流程。

### 2.3.10 硬件编码支持 roi 编码

参考 examples/multimedia/ff\_video\_encode/例子。设置 roi\_enable 既可启用 roi 编码。

Roi 编码数据通过 av\_frame side data 传递。

Roi 数据结构定义为

```

609 typedef union {
610     struct {
611         int mb_force_mode;
612         int mb_qp;
613     }H264;
614     struct {
615         int ctu_force_mode;
616         int ctu_coeff_drop;
617
618         int sub_ctu_qp_0;
619         int sub_ctu_qp_1;
620         int sub_ctu_qp_2;
621         int sub_ctu_qp_3;
622
623         int lambda_sad_0;
624         int lambda_sad_1;
625         int lambda_sad_2;
626         int lambda_sad_3;
627     }HEVC;
628 } RoiField;
629 } AVBMRoiInfo;
630
631 typedef struct AVBMRoiInfo {
632     // int numbers;
633     /* Enable ROI map. */
634     int customRoiMapEnable;
635     /* Enable custom lambda map. */
636     int customLambdaMapEnable;
637     /* Force CTU to be encoded with intra or to be skipped. */
638     int customModeMapEnable;
639     /* Force all coefficients to be zero after TQ or not for each CTU (to be dropped).*/
640     int customCoefDropEnable;
641
642     RoiField field[0x40000];
643 } AVBMRoiInfo;
644
```

字段说明:

- QP Map

H264 下 QP 以宏块 16x16 为单位给出。HEVC 下 QP 以 sub-ctu (32x32) 为单位给出。QP 对应的就是 video 编码中的 Qstep，取值范围为 0-51。

- Lamda Map

lamda 是用来控制和调节 IP 内部的 RC 计算公式

$$\text{cost} = \text{distortion} + \text{lamda} * \text{rate}$$

这个调节参数仅在 HEVC 下有效，允许以 32x32 sub-CTU 模块为单位控制。

- Mode Map

这个参数用来指定模式选择。0 –不适用 1 –skip mode 2- intra mode。H264 下以宏块 16x16 为单位控制，HEVC 下以 CTU 64x64 为单位控制。

- Zero-cut Flag

仅在 HEVC 下有效。将当前 CTU 64x64 残差系数全部置为 0，从而节省出更多的比特给其他更重要的部分。

## 2.4 SOPHGO LIBYUV 使用指南

### 2.4.1 简介

BM1686 系列芯片中的各种硬件模块，可以加速对图片和视频的处理。颜色转换方面，采用专用硬件来加速速度很快。

但在有些场合，也会存在一些专用硬件覆盖不到的特殊情况。此时采用经过 SIMD 加速优化的软件实现，成为专用硬件有力的补充。

SOPHGO 增强版 **libyuv**，是随同 SDK 一同发布的一个组件。目的是充分利用 BM1686 系列芯片提供的 8 核 A53 处理器，通过软件手段为硬件的局限性提供补充。

除了 libyuv 提供的标准函数之外，针对 AI 的需求，在 SOPHGO 增强版 libyuv 中，补充了 27 个扩展函数。

注意：这里说的是运行在 BM1686 系列的 A53 处理器上，而不是 host 的处理器。这从设备加速的角度是可以理解的。这样可以避免占用 host 的 CPU。

### 2.4.2 libyuv 扩展说明

新增了如下增强 AI 应用方面的 API。

#### **fast\_memcpy**

```
void* fast_memcpy(void *dst, const void *src, size_t n)
```

| 功能  | CPU SIMD 指令实现 memcpy 功能。从内存区域 src 拷贝 n 个字节到内存区域 dst |          |
|-----|-----------------------------------------------------|----------|
| 参数  | src                                                 | 源内存区域    |
|     | n                                                   | 需要拷贝的字节数 |
|     | dst                                                 | 目的内存区域   |
| 返回值 | 返回一个指向 dst 的指针                                      |          |

**RGB24ToI400**

```
int RGB24ToI400(const uint8_t* src_rgb24, int src_stride_rgb24, uint8_t*
dst_y, int dst_stride_y, int width, int height);
```

| 功能  | 将一帧 BGR 数据转换成 BT.601 灰度数据 |                             |
|-----|---------------------------|-----------------------------|
| 参数  | src_rgb24                 | packed BGR 图像数据所在的内存虚地址     |
|     | src_stride_rgb24          | 内存中每行 BGR 图像实际跨度            |
|     | dst_y                     | 灰度图像虚拟地址                    |
|     | dst_stride_y              | 内存中每行灰度图像实际跨度               |
|     | width                     | 每行 BGR 图像数据中 packed BGR 的数量 |
|     | height                    | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。       |                             |

**RAWToI400**

```
int RAWToI400(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y,
int dst_stride_y, int width, int height);
```

| 功能  | 将一帧 RGB 数据转换成 BT.601 灰度数据 |                             |
|-----|---------------------------|-----------------------------|
| 参数  | src_row                   | packed BGR 图像数据所在的内存虚地址     |
|     | src_stride_row            | 内存中每行 BGR 图像实际跨度            |
|     | dst_y                     | 灰度图像虚拟地址                    |
|     | dst_stride_y              | 内存中每行灰度图像实际跨度               |
|     | width                     | 每行 BGR 图像数据中 packed BGR 的数量 |
|     | height                    | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。       |                             |



**I400ToRGB24**

```
int I400ToRGB24(const uint8_t* src_y, int src_stride_y, uint8_t* dst_rgb24,
int dst_stride_rgb24, int width, int height);
```

| 功能  | 将一帧 BT.601 灰度数据转换成 BGR 数据 |                             |
|-----|---------------------------|-----------------------------|
| 参数  | src_y                     | 灰度图像虚拟地址                    |
|     | src_stride_y              | 内存中每行灰度图像实际跨度               |
|     | dst_rgb24                 | packed BGR 图像数据所在的内存虚地址     |
|     | dst_stride_rgb24          | 内存中每行 BGR 图像实际跨度            |
|     | width                     | 每行 BGR 图像数据中 packed BGR 的数量 |
|     | height                    | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。       |                             |

**I400ToRAW**

```
int I400ToRAW(const uint8_t* src_y, int src_stride_y, uint8_t* dst_raw, int
dst_stride_raw, int width, int height);
```

| 功能  | 将一帧 BT.601 灰度数据转换成 RGB 数据 |                             |
|-----|---------------------------|-----------------------------|
| 参数  | src_y                     | 灰度图像虚拟地址                    |
|     | src_stride_y              | 内存中每行灰度图像实际跨度               |
|     | dst_rgb24                 | packed RGB 图像数据所在的内存虚地址     |
|     | dst_stride_rgb24          | 内存中每行 RGB 图像实际跨度            |
|     | width                     | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                    | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。       |                             |

**J400ToRGB24**

```
int J400ToRGB24(const uint8_t* src_y, int src_stride_y, uint8_t* dst_rgb24,
int dst_stride_rgb24, int width, int height);
```

| 功能  | 将一帧 BT.601 full range 灰度数据转换成 BGR 数据 |                             |
|-----|--------------------------------------|-----------------------------|
| 参数  | src_y                                | 灰度图像虚拟地址                    |
|     | src_stride_y                         | 内存中每行灰度图像实际跨度               |
|     | dst_rgb24                            | packed BGR 图像数据所在的内存虚地址     |
|     | dst_stride_rgb24                     | 内存中每行 BGR 图像实际跨度            |
|     | width                                | 每行 BGR 图像数据中 packed BGR 的数量 |
|     | height                               | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                  |                             |

**RAWToJ400**

```
int RAWToJ400(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y,
int dst_stride_y, int width, int height);
```

| 功能  | 将一帧 RGB 数据转换成 BT.601 full range 灰度数据 |                             |
|-----|--------------------------------------|-----------------------------|
| 参数  | src_raw                              | packed RGB 图像数据所在的内存虚地址     |
|     | src_stride_raw                       | 内存中每行 RGB 图像实际跨度            |
|     | dst_y                                | 灰度图像虚拟地址                    |
|     | dst_stride_y                         | 内存中每行灰度图像实际跨度               |
|     | width                                | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                               | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                  |                             |

**J400ToRAW**

```
int J400ToRAW(const uint8_t* src_y, int src_stride_y, uint8_t* dst_raw, int
dst_stride_raw, int width, int height);
```

| 功能  | 将一帧 BT.601 full range 灰度数据转换成 RGB 数据 |                             |
|-----|--------------------------------------|-----------------------------|
| 参数  | src_y                                | 灰度图像虚拟地址                    |
|     | src_stride_y                         | 内存中每行灰度图像实际跨度               |
|     | dst_rgb24                            | packed RGB 图像数据所在的内存虚地址     |
|     | dst_stride_rgb24                     | 内存中每行 RGB 图像实际跨度            |
|     | width                                | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                               | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                  |                             |

**RAWToNV12**

```
int RAWToNV12(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y,
int dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);
```

| 功能  | 将一帧 RGB 数据转换成 BT.601 limited range 的 semi-planar YCbCr 420 数据 |                             |
|-----|---------------------------------------------------------------|-----------------------------|
| 参数  | src_raw                                                       | packed RGB 图像数据所在的内存虚地址     |
|     | src_stride_raw                                                | 内存中每行 RGB 图像实际跨度            |
|     | dst_y                                                         | Y 分量的虚拟地址                   |
|     | dst_stride_y                                                  | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_uv                                                        | CbCr 分量的虚拟地址                |
|     | dst_stride_uv                                                 | 内存中每行 CbCr 分量数据的实际跨度        |
|     | width                                                         | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                                        | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                                           |                             |

**RGB24ToNV12**

```
int RGB24ToNV12(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y,
int dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);
```

| 功能  | 将一帧 BGR 数据转换成 BT.601 limited range 的 semi-planar YCbCr 420 数据 |                             |
|-----|---------------------------------------------------------------|-----------------------------|
| 参数  | src_raw                                                       | packed BGR 图像数据所在的内存虚地址     |
|     | src_stride_raw                                                | 内存中每行 BGR 图像实际跨度            |
|     | dst_y                                                         | Y 分量的虚拟地址                   |
|     | dst_stride_y                                                  | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_uv                                                        | CbCr 分量的虚拟地址                |
|     | dst_stride_uv                                                 | 内存中每行 CbCr 分量数据的实际跨度        |
|     | width                                                         | 每行 BGR 图像数据中 packed BGR 的数量 |
|     | height                                                        | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                                           |                             |

**RAWToJ420**

```
int RAWToJ420(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y,
int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int
dst_stride_v, int width, int height);
```

| 功能  | 将一帧 RGB 数据转换成 BT.601 full range 的 semi-planar YCbCr 420 数据 |                             |
|-----|------------------------------------------------------------|-----------------------------|
| 参数  | src_raw                                                    | packed RGB 图像数据所在的内存虚地址     |
|     | src_stride_raw                                             | 内存中每行 RGB 图像实际跨度            |
|     | dst_y                                                      | Y 分量的虚拟地址                   |
|     | dst_stride_y                                               | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_u                                                      | Cb 分量的虚拟地址                  |
|     | dst_stride_u                                               | 内存中每行 Cb 分量数据的实际跨度          |
|     | dst_v                                                      | Cr 分量的虚拟地址                  |
|     | dst_stride_v                                               | 内存中每行 Cr 分量数据的实际跨度          |
|     | width                                                      | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                                     | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                                        |                             |

**J420ToRAW**

```
int J420ToRAW(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_raw, int
dst_stride_raw, int width, int height);
```

| 功能  | 将一帧 BT.601 full range 的 YCbCr 420 数据转换成 RGB 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_y                                          | Y 分量的虚拟地址                   |
|     | src_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | src_u                                          | Cb 分量的虚拟地址                  |
|     | src_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | src_v                                          | Cr 分量的虚拟地址                  |
|     | src_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | dst_raw                                        | packed RGB 图像数据所在的内存虚地址     |
|     | dst_stride_raw                                 | 内存中每行 RGB 图像数据的实际跨度         |
|     | width                                          | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                         | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**RAWToJ422**

```
int RAWToJ422(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y,
int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int
dst_stride_v, int width, int height);
```

| 功能  | 将一帧 RGB 数据转换成 BT.601 full range 的 YCbCr 422 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_raw                                        | packed RGB 图像数据所在的内存虚地址     |
|     | src_stride_raw                                 | 内存中每行 RGB 图像实际跨度            |
|     | dst_y                                          | Y 分量的虚拟地址                   |
|     | dst_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_u                                          | Cb 分量的虚拟地址                  |
|     | dst_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | dst_v                                          | Cr 分量的虚拟地址                  |
|     | dst_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | width                                          | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                         | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**J422ToRAW**

```
int J422ToRAW(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_raw, int
dst_stride_raw, int width, int height);
```

| 功能  | 将一帧 BT.601 full range 的 YCbCr 422 数据转换成 RGB 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_y                                          | Y 分量的虚拟地址                   |
|     | src_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | src_u                                          | Cb 分量的虚拟地址                  |
|     | src_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | src_v                                          | Cr 分量的虚拟地址                  |
|     | src_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | dst_raw                                        | packed RGB 图像数据所在的内存虚地址     |
|     | dst_stride_raw                                 | 内存中每行 RGB 图像数据的实际跨度         |
|     | width                                          | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                         | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**RGB24ToJ422**

```
int RGB24ToJ422(const uint8_t* src_rgb24, int src_stride_rgb24, uint8_t*
dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v,
int dst_stride_v, int width, int height);
```

| 功能  | 将一帧 BGR 数据转换成 BT.601 full range 的 YCbCr 422 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_rgb24                                      | packed BGR 图像数据所在的内存虚地址     |
|     | src_stride_rgb24                               | 内存中每行 BGR 图像实际跨度            |
|     | dst_y                                          | Y 分量的虚拟地址                   |
|     | dst_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_u                                          | Cb 分量的虚拟地址                  |
|     | dst_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | dst_v                                          | Cr 分量的虚拟地址                  |
|     | dst_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | width                                          | 每行 BGR 图像数据中 packed BGR 的数量 |
|     | height                                         | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**J422ToRGB24**

```
int J422ToRGB24(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_rgb24,
int dst_stride_rgb24, int width, int height);
```

| 功能  | 将一帧 BT.601 full range 的 YCbCr 422 数据转换成 BGR 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_y                                          | Y 分量的虚拟地址                   |
|     | src_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | src_u                                          | Cb 分量的虚拟地址                  |
|     | src_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | src_v                                          | Cr 分量的虚拟地址                  |
|     | src_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | dst_rgb24                                      | packed BGR 图像数据所在的内存虚地址     |
|     | dst_stride_rgb24                               | 内存中每行 BGR 图像数据的实际跨度         |
|     | width                                          | 每行 RGB 图像数据中 packed BGR 的数量 |
|     | height                                         | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**RAWToJ444**

```
int RAWToJ444(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y,
int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int
dst_stride_v, int width, int height);
```

| 功能  | 将一帧 RGB 数据转换成 BT.601 full range 的 YCbCr 444 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_raw                                        | packed RGB 图像数据所在的内存虚地址     |
|     | src_stride_raw                                 | 内存中每行 RGB 图像实际跨度            |
|     | dst_y                                          | Y 分量的虚拟地址                   |
|     | dst_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_u                                          | Cb 分量的虚拟地址                  |
|     | dst_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | dst_v                                          | Cr 分量的虚拟地址                  |
|     | dst_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | width                                          | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                         | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**J444ToRAW**

```
int J444ToRAW(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_raw, int
dst_stride_raw, int width, int height);
```

| 功能  | 将一帧 BT.601 full range 的 YCbCr 444 数据转换成 RGB 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_y                                          | Y 分量的虚拟地址                   |
|     | src_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | src_u                                          | Cb 分量的虚拟地址                  |
|     | src_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | src_v                                          | Cr 分量的虚拟地址                  |
|     | src_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | dst_raw                                        | packed RGB 图像数据所在的内存虚地址     |
|     | dst_stride_raw                                 | 内存中每行 RGB 图像数据的实际跨度         |
|     | width                                          | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                         | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**RGB24ToJ444**

```
int RGB24ToJ444(const uint8_t* src_rgb24, int src_stride_rgb24, uint8_t*
dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v,
int dst_stride_v, int width, int height);
```

| 功能  | 将一帧 BGR 数据转换成 BT.601 full range 的 YCbCr 444 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_rgb24                                      | packed BGR 图像数据所在的内存虚地址     |
|     | src_stride_rgb24                               | 内存中每行 BGR 图像实际跨度            |
|     | dst_y                                          | Y 分量的虚拟地址                   |
|     | dst_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_u                                          | Cb 分量的虚拟地址                  |
|     | dst_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | dst_v                                          | Cr 分量的虚拟地址                  |
|     | dst_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | width                                          | 每行 BGR 图像数据中 packed BGR 的数量 |
|     | height                                         | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |

**J444ToRGB24**

```
int J444ToRGB24(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_rgb24,
int dst_stride_rgb24, int width, int height);
```

| 功能  | 将一帧 BT.601 full range 的 YCbCr 444 数据转换成 BGR 数据 |                             |
|-----|------------------------------------------------|-----------------------------|
| 参数  | src_y                                          | Y 分量的虚拟地址                   |
|     | src_stride_y                                   | 内存中每行 Y 分量数据的实际跨度           |
|     | src_u                                          | Cb 分量的虚拟地址                  |
|     | src_stride_u                                   | 内存中每行 Cb 分量数据的实际跨度          |
|     | src_v                                          | Cr 分量的虚拟地址                  |
|     | src_stride_v                                   | 内存中每行 Cr 分量数据的实际跨度          |
|     | dst_rgb24                                      | packed BGR 图像数据所在的内存虚地址     |
|     | dst_stride_rgb24                               | 内存中每行 BGR 图像数据的实际跨度         |
|     | width                                          | 每行 RGB 图像数据中 packed BGR 的数量 |
|     | height                                         | BGR 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                            |                             |



**H420ToJ420**

```
int H420ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y,
int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int
dst_stride_v, int width, int height);
```

| 功<br>能      | 将一帧 BT.709 limited range 的 YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用 |                             |
|-------------|-----------------------------------------------------------------------------------------|-----------------------------|
| 参<br>数      | src_y                                                                                   | Y 分量的虚拟地址                   |
|             | src_stride_y                                                                            | 内存中每行 Y 分量数据的实际跨度           |
|             | src_u                                                                                   | Cb 分量的虚拟地址                  |
|             | src_stride_u                                                                            | 内存中每行 Cb 分量数据的实际跨度          |
|             | src_v                                                                                   | Cr 分量的虚拟地址                  |
|             | src_stride_v                                                                            | 内存中每行 Cr 分量数据的实际跨度          |
|             | dst_y                                                                                   | Y 分量的虚拟地址                   |
|             | dst_stride_y                                                                            | 内存中每行 Y 分量数据的实际跨度           |
|             | dst_u                                                                                   | Cb 分量的虚拟地址                  |
|             | dst_stride_u                                                                            | 内存中每行 Cb 分量数据的实际跨度          |
|             | dst_v                                                                                   | Cr 分量的虚拟地址                  |
|             | dst_stride_v                                                                            | 内存中每行 Cr 分量数据的实际跨度          |
|             | width                                                                                   | 每行 RGB 图像数据中 packed RGB 的数量 |
|             | height                                                                                  | RGB 图像数据的有效行数               |
| 返<br>回<br>值 | 0, 正常结束; 非 0, 参数异常。                                                                     |                             |

**I420ToJ420**

```
int I420ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y,
int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int
dst_stride_v, int width, int height);
```

| 功<br>能      | 将一帧 BT.601 limited range 的 YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用 |                             |
|-------------|-----------------------------------------------------------------------------------------|-----------------------------|
| 参<br>数      | src_y                                                                                   | Y 分量的虚拟地址                   |
|             | src_stride_y                                                                            | 内存中每行 Y 分量数据的实际跨度           |
|             | src_u                                                                                   | Cb 分量的虚拟地址                  |
|             | src_stride_u                                                                            | 内存中每行 Cb 分量数据的实际跨度          |
|             | src_v                                                                                   | Cr 分量的虚拟地址                  |
|             | src_stride_v                                                                            | 内存中每行 Cr 分量数据的实际跨度          |
|             | dst_y                                                                                   | Y 分量的虚拟地址                   |
|             | dst_stride_y                                                                            | 内存中每行 Y 分量数据的实际跨度           |
|             | dst_u                                                                                   | Cb 分量的虚拟地址                  |
|             | dst_stride_u                                                                            | 内存中每行 Cb 分量数据的实际跨度          |
|             | dst_v                                                                                   | Cr 分量的虚拟地址                  |
|             | dst_stride_v                                                                            | 内存中每行 Cr 分量数据的实际跨度          |
|             | width                                                                                   | 每行 RGB 图像数据中 packed RGB 的数量 |
|             | height                                                                                  | RGB 图像数据的有效行数               |
| 返<br>回<br>值 | 0, 正常结束; 非 0, 参数异常。                                                                     |                             |

**NV12ToJ420**

```
int NV12ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_uv,
int src_stride_uv, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int
dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);
```

| 功能  | 将一帧 BT.601 limited range 的 semi-plannar YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用 |                             |
|-----|------------------------------------------------------------------------------------------------------|-----------------------------|
| 参数  | src_y                                                                                                | Y 分量的虚拟地址                   |
|     | src_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度           |
|     | src_uv                                                                                               | CbCr 分量的虚拟地址                |
|     | src_stride_uv                                                                                        | 内存中每行 CbCr 分量数据的实际跨度        |
|     | dst_y                                                                                                | Y 分量的虚拟地址                   |
|     | dst_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_u                                                                                                | Cb 分量的虚拟地址                  |
|     | dst_stride_u                                                                                         | 内存中每行 Cb 分量数据的实际跨度          |
|     | dst_v                                                                                                | Cr 分量的虚拟地址                  |
|     | dst_stride_v                                                                                         | 内存中每行 Cr 分量数据的实际跨度          |
|     | width                                                                                                | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                                                                               | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                                                                                  |                             |

**NV21ToJ420**

```
int NV21ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_vu,
int src_stride_vu, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int
dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);
```

| 功能  | 将一帧 BT.601 limited range 的 semi-plannar YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用 |                             |
|-----|------------------------------------------------------------------------------------------------------|-----------------------------|
| 参数  | src_y                                                                                                | Y 分量的虚拟地址                   |
|     | src_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度           |
|     | src_vu                                                                                               | CrCb 分量的虚拟地址                |
|     | src_stride_vu                                                                                        | 内存中每行 CrCb 分量数据的实际跨度        |
|     | dst_y                                                                                                | Y 分量的虚拟地址                   |
|     | dst_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度           |
|     | dst_u                                                                                                | Cb 分量的虚拟地址                  |
|     | dst_stride_u                                                                                         | 内存中每行 Cb 分量数据的实际跨度          |
|     | dst_v                                                                                                | Cr 分量的虚拟地址                  |
|     | dst_stride_v                                                                                         | 内存中每行 Cr 分量数据的实际跨度          |
|     | width                                                                                                | 每行 RGB 图像数据中 packed RGB 的数量 |
|     | height                                                                                               | RGB 图像数据的有效行数               |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                                                                                  |                             |

**I444ToNV12**

```
int I444ToNV12(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y, int
dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);
```

| 功能  | 将一帧 YCbCr 444 数据转换成 semi-planar YCbCr 420 数据。可用于 full range , 也可以用于 limited range 的数据。不涉及颜色空间转换, 可灵活使用 |                      |
|-----|--------------------------------------------------------------------------------------------------------|----------------------|
| 参数  | src_y                                                                                                  | 源图像 Y 分量的虚拟地址        |
|     | src_stride_y                                                                                           | 内存中每行 Y 分量数据的实际跨度    |
|     | src_u                                                                                                  | 源图像 Cb 分量的虚拟地址       |
|     | src_stride_u                                                                                           | 内存中每行 Cb 分量数据的实际跨度   |
|     | src_v                                                                                                  | 源图像 Cr 分量的虚拟地址       |
|     | src_stride_v                                                                                           | 内存中每行 Cr 分量数据的实际跨度   |
|     | dst_y                                                                                                  | 目的图像 Y 分量的虚拟地址       |
|     | dst_stride_y                                                                                           | 内存中每行 Y 分量数据的实际跨度    |
|     | dst_uv                                                                                                 | 目的图像 CbCr 分量的虚拟地址    |
|     | dst_stride_uv                                                                                          | 内存中每行 CbCr 分量数据的实际跨度 |
|     | width                                                                                                  | 每行图像数据中像素的数量         |
|     | height                                                                                                 | 图像数据像素的有效行数          |
| 返回值 | 0, 正常结束; 非 0, 参数异常。                                                                                    |                      |

**I422ToNV12**

```
int I422ToNV12(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u,
int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y, int
dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);
```

|            |                                                                                                      |                      |
|------------|------------------------------------------------------------------------------------------------------|----------------------|
| <b>功能</b>  | 将一帧 YCbCr 422 数据转换成 semi-plannar YCbCr 420 数据。可用于 full range，也可以用于 limited range 的数据。不涉及颜色空间转换，可灵活使用 |                      |
| <b>参数</b>  | src_y                                                                                                | 源图像 Y 分量的虚拟地址        |
|            | src_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度    |
|            | src_u                                                                                                | 源图像 Cb 分量的虚拟地址       |
|            | src_stride_u                                                                                         | 内存中每行 Cb 分量数据的实际跨度   |
|            | src_v                                                                                                | 源图像 Cr 分量的虚拟地址       |
|            | src_stride_v                                                                                         | 内存中每行 Cr 分量数据的实际跨度   |
|            | dst_y                                                                                                | 目的图像 Y 分量的虚拟地址       |
|            | dst_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度    |
|            | dst_uv                                                                                               | 目的图像 CbCr 分量的虚拟地址    |
|            | dst_stride_uv                                                                                        | 内存中每行 CbCr 分量数据的实际跨度 |
|            | width                                                                                                | 每行图像数据中像素的数量         |
|            | height                                                                                               | 图像数据像素的有效行数          |
| <b>返回值</b> | 0, 正常结束; 非 0, 参数异常。                                                                                  |                      |

#### I400ToNV12

```
int I400ToNV12(const uint8_t* src_y, int src_stride_y, uint8_t* dst_y, int
dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);
```

|            |                                                                                                      |                      |
|------------|------------------------------------------------------------------------------------------------------|----------------------|
| <b>功能</b>  | 将一帧 YCbCr 400 数据转换成 semi-plannar YCbCr 420 数据。可用于 full range，也可以用于 limited range 的数据。不涉及颜色空间转换，可灵活使用 |                      |
| <b>参数</b>  | src_y                                                                                                | 源图像 Y 分量的虚拟地址        |
|            | src_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度    |
|            | dst_y                                                                                                | 目的图像 Y 分量的虚拟地址       |
|            | dst_stride_y                                                                                         | 内存中每行 Y 分量数据的实际跨度    |
|            | dst_uv                                                                                               | 目的图像 CbCr 分量的虚拟地址    |
|            | dst_stride_uv                                                                                        | 内存中每行 CbCr 分量数据的实际跨度 |
|            | width                                                                                                | 每行图像数据中像素的数量         |
|            | height                                                                                               | 图像数据像素的有效行数          |
| <b>返回值</b> | 0, 正常结束; 非 0, 参数异常。                                                                                  |                      |

## 2.5 SOPHGO JPEG 使用指南

### 2.5.1 简介

JPEG 在 BM1688 产品系列中是一个可以实现多种图片格式编解码的功能模块。该模块包含编码、解码、镜像、旋转及 ROI 等功能，支持各种 YUV 格式。可以支持的分辨率范围为 16x16~32768x32768，通过硬件加速，编解码性能也十分出色。

### 2.5.2 JPEG 数据结构说明

#### BmJpuColorFormat

该类型表示图像的像素格式。

**BmJpuColorFormat** 类型定义如下：

```
typedef enum
{
    /* planar 4:2:0; if the chroma_interleave parameter is 1, the corresponding format
    ↪ is NV12, otherwise it is I420 */
    BM_JPU_COLOR_FORMAT_YUV420 = 0,
    /* planar 4:2:2; if the chroma_interleave parameter is 1, the corresponding format
    ↪ is NV16 */
    BM_JPU_COLOR_FORMAT_YUV422_HORIZONTAL = 1,
    /* 4:2:2 vertical, actually 2:2:4 (according to the JPU docs); no corresponding
    ↪ format known for the chroma_interleave=1 case */
    /* NOTE: this format is rarely used, and has only been seen in a few JPEG files */
    BM_JPU_COLOR_FORMAT_YUV422_VERTICAL = 2,
    /* planar 4:4:4; if the chroma_interleave parameter is 1, the corresponding format
    ↪ is NV24 */
    BM_JPU_COLOR_FORMAT_YUV444 = 3,
    /* 8-bit greyscale */
    BM_JPU_COLOR_FORMAT_YUV400 = 4,
    /* RGBP */
    BM_JPU_COLOR_FORMAT_RGB = 5,
    /* BUTT */
    BM_JPU_COLOR_FORMAT_BUTT
} BmJpuColorFormat;
```

- **BM\_JPU\_COLOR\_FORMAT\_YUV420**  
YUV420 格式。在 cbr\_interleave=1 的时候对应 NV12 格式。
- **BM\_JPU\_COLOR\_FORMAT\_YUV422\_HORIZONTAL**  
YUV422 格式。在 cbr\_interleave=1 的时候对应 NV16 格式。
- **BM\_JPU\_COLOR\_FORMAT\_YUV422\_VERTICAL**  
YUV422 格式（实际按 2: 2: 4 排列），JPU 定义的一种格式，不常用。

- **BM\_JPU\_COLOR\_FORMAT\_YUV444**  
YUV444 格式。在 cbc\_r\_interleave=1 的时候对应 NV24 格式。
- **BM\_JPU\_COLOR\_FORMAT\_YUV400**  
YUV400 格式。对应灰度图，只有 Y 分量。
- **BM\_JPU\_COLOR\_FORMAT\_RGB**  
RGB 格式，暂未使用。
- **BM\_JPU\_COLOR\_FORMAT\_BUTT**  
未定义格式，用于设置默认值和异常判断。

### BmJpuFramebuffer

该结构体用于在 BMAPI 中描述一帧 YUV 数据。

**BmJpuFramebuffer** 结构体定义如下：

```
typedef struct {
    unsigned int y_stride;
    unsigned int cbc_r_stride;
    bm_device_mem_t *dma_buffer;
    size_t y_offset;
    size_t cb_offset;
    size_t cr_offset;

    void *context;
    int already_marked;
    void *internal;
} BmJpuFramebuffer;
```

- **y\_stride**  
亮度（Y）分量的步长。
- **cbc\_r\_stride**  
色度（Cb、Cr）分量的步长。
- **dma\_buffer**  
用于存放 YUV 数据的一块设备内存，由 bmlib 分配。
- **y\_offset**  
Y 分量起始地址相对于 dma\_buffer 中物理地址的偏移量。
- **cb\_offset**  
Cb 分量起始地址相对于 dma\_buffer 中物理地址的偏移量。

- **cr\_offset**  
Cr 分量起始地址相对于 dma\_buffer 中物理地址的偏移量。
- **context**  
用于保存解码上下文信息。
- **already\_marked**  
标记当前帧是否可以输出。
- **internal**  
内部数据，由用户决定如何使用。

### BmJpuFramebufferSizes

该结构体用于记录对齐之后的 framebuffer 信息

**BmJpuFramebufferSizes** 结构体定义如下：

```
typedef struct{
    unsigned int aligned_frame_width, aligned_frame_height;
    unsigned int y_stride, cbcr_stride;
    unsigned int y_size, cbcr_size;
    unsigned int total_size;
    int chroma_interleave;
} BmJpuFramebufferSizes;
```

- **aligned\_frame\_width**  
对齐之后的宽度。
- **aligned\_frame\_height**  
对齐之后的高度。
- **y\_stride**  
亮度（Y）分量的步长。
- **cbcr\_stride**  
色度（Cb、Cr）分量的步长。
- **y\_size**  
Y 分量的总数据量，单位：byte。
- **cbcr\_size**  
Cb、Cr 分量的总数据量，单位：byte。
- **total\_size**  
framebuffer 的总数据量，单位：byte。



- **chroma\_interleave**

Cb、Cr 分量是否交错排列。

### BmJpuDecOpenParams

该结构体用于定义打开解码器的参数，换句话说就是设置解码器的属性（接收图片的大小，解码缓存区大小等属性）。

**BmJpuDecOpenParams** 结构体定义如下：

```
typedef struct
{
    /* These are necessary with some formats which do not store the width
    * and height in the bitstream. If the format does store them, these
    * values can be set to zero. */
    unsigned int frame_width;
    unsigned int frame_height;

    BmJpuColorFormat color_format;
    /* If this is 1, then Cb and Cr are interleaved in one shared chroma
    * plane, otherwise they are separated in their own planes.
    * See the BmJpuColorFormat documentation for the consequences of this. */
    int chroma_interleave;

    /* 0: no scaling; n(1-3): scale by 2^n; */
    unsigned int scale_ratio;

    /* The DMA buffer size for bitstream */
    int bs_buffer_size;
#ifdef _WIN32
    uint8_t *buffer;
#else
    uint8_t *buffer __attribute__((deprecated));
#endif

    int device_index;

    int rotationEnable;
    int mirrorEnable;
    int mirrorDirection;
    int rotationAngle;

    int roiEnable;
    int roiWidth;
    int roiHeight;
    int roiOffsetX;
    int roiOffsetY;

    int framebuffer_recycle;
    size_t framebuffer_size;
} BmJpuDecOpenParams;
```

- **frame\_width**  
输入图像的宽度。
- **frame\_height**  
输入图像的高度。
- **color\_format**  
输入图像的编码格式。
- **chroma\_interleave**  
色度分量的存储方式的标识选项，可以是交错存储也可以是分开存储。
- **scale\_ratio**  
用于指定视频解码时的缩放比例。它决定了图像在解码过程中是否进行大小调整（即缩放）。如果值为 0，则表示不进行任何缩放；如果值在 1 到 3 之间（包括 1 和 3），则表示将图像按照 2 的 n 次幂进行缩放，其中 n 为 scale\_ratio 的值。
- **bs\_buffer\_size**  
表示码流的缓冲区的大小，这里记录了存储输入图片需要的字节大小。
- **buffer**  
用于存储输入图片的具体内容。（已弃用）
- **device\_index**  
解码设备 ID。
- **rotationEnable**  
是否做旋转操作的标识，0 表示不旋转，1 表示旋转。
- **mirrorEnable**  
是否做镜像操作的标识，0 表示不镜像，1 表示镜像。
- **mirrorDirection**  
表示镜像的方向，可选项：0 - 水平，1 - 竖直。
- **rotationAngle**  
表示旋转的角度，可选项：1 - 90°、2 - 180°、3 - 270°。
- **roiEnable**  
是否设置 ROI（感兴趣区域）。
- **roiWidth**  
ROI 的宽度。
- **roiHeight**  
ROI 的高度。

- **roiOffsetX**  
ROI 相对图像左上角的水平偏移量。
- **roiOffsetY**  
ROI 相对图像左上角的垂直偏移量。
- **framebuffer\_recycle**  
是否复用 framebuffer。
- **framebuffer\_size**  
如果复用 framebuffer，设置 framebuffer 的大小，单位：byte。

### FramebufferList

该结构体用于定义存放 framebuffer 的链表。

**FramebufferList** 结构体定义如下：

```
typedef struct FramebufferList {  
    struct FramebufferList *next;  
    BmJpuFramebuffer *fb;  
    int chn_id;  
    int chn_fd;  
} FramebufferList;
```

- **next**  
表示链表的下一个节点。
- **fb**  
表示一个 framebuffer。
- **chn\_id**  
表示 framebuffer 对应的通道 ID。
- **chn\_fd**  
表示 framebuffer 对应的通道句柄。

### BmJpuDecoder

该结构体定义了 BMAPI 内部的解码器，包含解码器句柄、设备 ID、解码分配的 framebuffer 等信息。

**BmJpuDecoder** 结构体定义如下：

```

struct _BmJpuDecoder
{
    unsigned int device_index;
    DecHandle handle;
    bm_device_mem_t *bs_dma_buffer;
    uint8_t *bs_virt_addr;
    uint64_t bs_phys_addr;
    int chroma_interleave;
    int scale_ratio;
    unsigned int old_jpeg_width;
    unsigned int old_jpeg_height;
    BmJpuColorFormat old_jpeg_color_format;
    unsigned int num_framebuffers, num_used_framebuffers;
    FrameBuffer *internal_framebuffers;
    BmJpuFramebuffer *framebuffers;
    BmJpuDecFrameEntry *frame_entries;
    DecInitialInfo initial_info;
    int initial_info_available;
    DecOutputInfo dec_output_info;
    int available_decoded_frame_idx;
    bm_jpu_dec_new_initial_info_callback initial_info_callback;
    void *callback_user_data;
    int framebuffer_recycle;
    int channel_id;
    FramebufferList *fb_list_head;
    FramebufferList *fb_list_curr;
};
typedef struct _BmJpuDecoder BmJpuDecoder;

```

- **device\_index**  
解码设备 ID。
- **handle**  
解码器句柄。
- **bs\_dma\_buffer**  
用于存放待解码码流的一块设备内存，由 bmlib 分配。
- **bs\_virt\_addr**  
存放待解码码流的虚拟地址。
- **bs\_phys\_addr**  
存放待解码码流的物理地址。
- **chroma\_interleave**  
表示色度分量是否交叉排列。
- **scale\_ratio**  
表示图像缩放比例。该参数同时控制图像在水平和垂直方向上的缩放级别。

- **old\_jpeg\_width**  
表示上一帧解码图像的宽度。
- **old\_jpeg\_height**  
表示上一帧解码图像的高度。
- **old\_jpeg\_color\_format**  
表示上一帧解码图像的编码格式。
- **num\_framebuffers**  
表示已分配的解码 buffer 数量。
- **num\_used\_framebuffers**  
表示已使用的解码 buffer 数量。
- **internal\_framebuffers**  
解码器内部注册到 JPU 的 framebuffer, 用于和 JPU 交互。
- **framebuffers**  
解码器内部分配的全部 framebuffer。
- **frame\_entries**  
解码器内部解码 buffer 控制结构, 指定了当前帧的 PTS (显示时间)、DTS (解码时间) 及使用状态。
- **initial\_info**  
用于记录解码器的一些初始配置。
- **initial\_info\_available**  
用来标识解码器是否完成初始化。
- **dec\_output\_info**  
用于记录解码器的输出信息。
- **available\_decoded\_frame\_idx**  
表示当前使用的解码帧序号。
- **initial\_info\_callback**  
一个用于初始化 framebuffer 的回调函数, 当解码器输入码流变化时会重新分配 framebuffer。
- **callback\_user\_data**  
用于存储上述回调函数中传递的用户自定义数据。
- **framebuffer\_recycle**

用于表示 framebuffer 是否复用的标识。在复用的情况下，可用同一个解码器实例解码不同分辨率、格式的码流。

- **channel\_id**

表示解码器通道 ID。

- **fb\_list\_head**

表示 framebuffer 链表中的头节点。解码器中使用链表存储当前使用中的 framebuffer，会在执行 flush 或 close 操作时全部释放，也可以使用 **bm\_jpu\_jpeg\_dec\_frame\_finished** 接口释放某一帧。

- **fb\_list\_curr**

表示 framebuffer 链表中的当前节点。用于将新生成的 framebuffer 加入链表。

## BmJpuJPEGDecoder

该结构体定义了对外提供的解码器，包含解码器句柄、设备 ID、解码分配的 framebuffer 等信息。

**BmJpuJPEGDecoder** 结构体定义如下：

```
typedef struct _BMJpuJPEGDecoder
{
    BmJpuDecoder *decoder;

    bm_device_mem_t *bitstream_buffer;
    size_t bitstream_buffer_size;
    unsigned int bitstream_buffer_alignment;

    BmJpuDecInitialInfo initial_info;

    BmJpuFramebuffer *framebuffers;
    bm_device_mem_t *fb_dmabuffers;
    unsigned int num_framebuffers;
    unsigned int num_extra_framebuffers;
    BmJpuFramebufferSizes calculated_sizes;

    BmJpuRawFrame raw_frame;
    int device_index;

    BmJpuFramebuffer *cur_framebuffer;
    bm_device_mem_t *cur_dma_buffer;
    void *opaque;

    int rotationEnable;
    int mirrorEnable;
    int mirrorDirection;
    int rotationAngle;
}
```

(续下页)

(接上页)

```

    int framebuffer_recycle;
    size_t framebuffer_size;
}BmJpuJPEGDecoder;

```

- **decoder**

BMAPI 内部定义的解码器。

- **bitstream\_buffer**

用于存放待解码码流的一块设备内存，由 bmlib 分配。

- **bitstream\_buffer\_size**

表示上述码流的内存大小，单位：byte。

- **bitstream\_buffer\_alignment**

表示上述码流的对齐要求，单位：byte。

- **initial\_info**

用于记录解码器的一些初始配置。

- **framebuffers**

用于记录解码器中 framebuffer 的地址及大小。

- **fb\_dmabuffers**

用于存储解码器中 framebuffer 的设备内存，由 bmlib 分配。

- **num\_framebuffers**

解码需要的 framebuffer 总帧数。

- **num\_extra\_framebuffers**

解码需要的 framebuffer 额外帧数，通常为 0。

- **calculated\_sizes**

记录对齐后的 framebuffer 大小信息。

- **raw\_frame**

表示原始帧数据，用于存储图像的原始数据和时间戳。

- **device\_index**

表示解码设备 ID。

- **cur\_framebuffer**

当前帧解码使用的 framebuffer，用于 framebuffer\_recycle 模式。

- **cur\_dma\_buffer**

当前帧解码使用的设备内存，用于 framebuffer\_recycle 模式。

- **opaque**  
未定义数据，由用户决定如何使用。
- **rotationEnable**  
是否做旋转操作的标识，0 表示不旋转，1 表示旋转。
- **mirrorEnable**  
是否做镜像操作的标识，0 表示不镜像，1 表示镜像。
- **mirrorDirection**  
表示镜像的方向，可选项：0 - 水平，1 - 竖直。
- **rotationAngle**  
表示旋转的角度，可选项：1 - 90°、2 - 180°、3 - 270°。
- **framebuffer\_recycle**  
用于表示 framebuffer 是否复用的标识。在复用的情况下，可用同一个解码器实例解码不同分辨率、格式的码流。
- **framebuffer\_size**  
framebuffer\_recycle 模式下需要申请的 framebuffer 内存大小，单位：byte。

### BmJpuJPEGDecInfo

该结构体记录了解码后的 YUV 数据信息，用于用户获取解码数据。

**BmJpuJPEGDecInfo** 结构体定义如下：

```
typedef struct
{
    /* Width and height of JPU framebuffers are aligned to internal boundaries.
    * The frame consists of the actual image pixels and extra padding pixels.
    * aligned_frame_width / aligned_frame_height specify the full width/height
    * including the padding pixels, and actual_frame_width / actual_frame_height
    * specify the width/height without padding pixels. */
    unsigned int aligned_frame_width, aligned_frame_height;
    unsigned int actual_frame_width, actual_frame_height;

    /* Stride and size of the Y, Cr, and Cb planes. The Cr and Cb planes always
    * have the same stride and size. */
    unsigned int y_stride, cbcr_stride;
    unsigned int y_size, cbcr_size;

    /* Offset from the start of a framebuffer's memory, in bytes. Note that the
    * Cb and Cr offset values are *not* the same, unlike the stride and size ones. */
    unsigned int y_offset, cb_offset, cr_offset;

    /* Framebuffer containing the pixels of the decoded frame. */
}
```

(续下页)



(接上页)

```

BmJpuFramebuffer *framebuffer;

/* Color format of the decoded frame. */
BmJpuColorFormat color_format;

int chroma_interleave;

int framebuffer_recycle;
size_t framebuffer_size;
} BmJpuJPEGDecInfo;

```

- **aligned\_frame\_width**  
对齐之后该帧数据的宽度。
- **aligned\_frame\_height**  
对齐之后该帧数据的高度。
- **actual\_frame\_width**  
原始图像的宽度。
- **actual\_frame\_height**  
原始图像的高度。
- **y\_stride**  
Y 分量的步长。
- **cbcr\_stride**  
Cb、Cr 分量的步长。
- **y\_size**  
Y 分量的总数据量，单位：byte。
- **cbcr\_size**  
Cb、Cr 分量的总数据量，单位：byte。
- **y\_offset**  
Y 分量起始地址相对于 framebuffer 中物理地址的偏移量。
- **cb\_offset**  
Cb 分量起始地址相对于 framebuffer 中物理地址的偏移量。
- **cr\_offset**  
Cr 分量起始地址相对于 framebuffer 中物理地址的偏移量。
- **framebuffer**  
用来记录解码后的 YUV 数据相关信息。

- **color\_format**  
YUV 数据编码格式。
- **chroma\_interleave**  
色度分量是否交错排列。
- **framebuffer\_recycle**  
用于表示 framebuffer 是否复用的标识。在复用的情况下，可用同一个解码器实例解码不同分辨率、格式的码流。
- **framebuffer\_size**  
framebuffer\_recycle 模式下需要申请的 framebuffer 内存大小，单位：byte。

### BmJpuEncoder

该结构体定义了 BMAPI 内部的编码器，包含编码器句柄、设备 ID、输出码流等信息。

**BmJpuEncoder** 结构体定义如下：

```
typedef struct _BmJpuEncoder
{
    unsigned int device_index;
    EncHandle handle;

    bm_device_mem_t *bs_dma_buffer;
    uint8_t *bs_virt_addr;

    BmJpuColorFormat color_format;
    unsigned int frame_width, frame_height;

    BmJpuFramebuffer *framebuffers;
} BmJpuEncoder;
```

- **device\_index**  
编码设备 ID。
- **handle**  
编码器句柄。
- **bs\_dma\_buffer**  
用于存放输出码流的一块设备内存，由 bmlib 分配。
- **bs\_virt\_addr**  
存放输出码流的虚拟地址。
- **color\_format**  
输出码流的编码格式。

- **frame\_width**  
输出码流的宽度。
- **frame\_height**  
输出码流的高度。
- **framebuffers**  
编码器内部使用的 framebuffer。

### BmJpuJPEGEncoder

该结构体定义了对外提供的编码器，包含编码器句柄、设备 ID、输出码流等信息。

**BmJpuJPEGEncoder** 结构体定义如下：

```
typedef struct _BmJpuJPEGEncoder
{
    BmJpuEncoder *encoder;

    bm_device_mem_t *bitstream_buffer;

    size_t bitstream_buffer_size;
    unsigned int bitstream_buffer_alignment;

    BmJpuEncInitialInfo initial_info;

    unsigned int frame_width, frame_height;

    BmJpuFramebufferSizes calculated_sizes;

    unsigned int quality_factor;

    BmJpuColorFormat color_format;
    int packed_format;
    int chroma_interleave;
    int device_index;

    int rotationEnable;
    int mirrorEnable;
    int mirrorDirection;
    int rotationAngle;
} BmJpuJPEGEncoder;
```

- **encoder**  
BMAPI 内部定义的编码器。
- **bitstream\_buffer**  
用于存放输出码流的一块设备内存，由 bmlib 分配。

- **bitstream\_buffer\_size**  
表示上述码流的内存大小，单位：byte。
- **bitstream\_buffer\_alignment**  
表示上述码流的对齐要求，单位：byte。
- **initial\_info**  
用于记录编码器的 framebuffer 初始配置。
- **frame\_width**  
输入图像的宽度。
- **frame\_height**  
输入图像的高度。
- **calculated\_sizes**  
记录对齐后的 framebuffer 大小信息。
- **quality\_factor**  
编码质量。
- **color\_format**  
输入图像的编码格式。
- **packed\_format**  
输入图像的打包格式。
- **chroma\_interleave**  
色度分量是否交错排列。
- **device\_index**  
表示编码设备 ID。
- **rotationEnable**  
是否做旋转操作的标识，0 表示不旋转，1 表示旋转。
- **mirrorEnable**  
是否做镜像操作的标识，0 表示不镜像，1 表示镜像。
- **mirrorDirection**  
表示镜像的方向，可选项：0 - 水平，1 - 竖直。
- **rotationAngle**  
表示旋转的角度，可选项：1 - 90°、2 - 180°、3 - 270°。

**BmJpuJPEGEncParams**

该结构体定义了编码配置参数及可配置的获取输出数据的接口函数。

**BmJpuJPEGEncParams** 结构体定义如下:

```
typedef struct
{
    /* Frame width and height of the input frame. These are the actual sizes;
    * they will be aligned internally if necessary. These sizes must not be
    * zero. */
    unsigned int frame_width, frame_height;

    /* Quality factor for JPEG encoding. 1 = best compression, 100 = best quality.
    * This is the exact same quality factor as used by libjpeg. */
    unsigned int quality_factor;

    /* Color format of the input frame. */
    BmJpuColorFormat color_format;

    /* Functions for acquiring and finishing output buffers. See the
    * typedef documentations in bmjpuapi.h for details about how
    * these functions should behave. */
    BmJpuEncAcquireOutputBuffer acquire_output_buffer;
    BmJpuEncFinishOutputBuffer finish_output_buffer;

    /* Function for directly passing the output data to the user
    * without copying it first.
    * Using this function will inhibit calls to acquire_output_buffer
    * and finish_output_buffer. */
    BmJpuWriteOutputData write_output_data;

    /* User supplied value that will be passed to the functions:
    * acquire_output_buffer, finish_output_buffer, write_output_data */
    void *output_buffer_context;

    int packed_format;
    int chroma_interleave;

    int rotationEnable;
    int mirrorEnable;
    int mirrorDirection;
    int rotationAngle;
} BmJpuJPEGEncParams;
```

- **frame\_width**  
输出码流的宽度。
- **frame\_height**  
输出码流的高度。
- **quality\_factor**

编码质量，可选 1（压缩率最高）~100（图像质量最好）。

- **color\_format**  
输出码流的编码格式。
- **acquire\_output\_buffer**  
用来获取编码码流输出 buffer 的回调函数。
- **finish\_output\_buffer**  
用来释放上述 buffer 的回调函数。
- **write\_output\_data**  
用来指定编码码流输出方式的回调函数，如：写入文件或写入指定的内存地址，与上述两个接口互斥。
- **output\_buffer\_context**  
用来保存输出数据的上下文。
- **packed\_format**  
输出码流的打包格式。
- **chroma\_interleave**  
色度分量是否交错排列
- **rotationEnable**  
是否做旋转操作的标识，0 表示不旋转，1 表示旋转。
- **mirrorEnable**  
是否做镜像操作的标识，0 表示不镜像，1 表示镜像。
- **mirrorDirection**  
表示镜像的方向，可选项：0 - 水平，1 - 竖直。
- **rotationAngle**  
表示旋转的角度，可选项：1 - 90°、2 - 180°、3 - 270°。

### 2.5.3 JPEG 接口说明

#### bm\_jpu\_dec\_load

该接口根据传入的 ID 打开指定的解码设备节点，可以通过 bmlib 管理内存分配。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_dec_load(int device_index);
```

参数说明：

- **device\_index**

解码设备 ID。

#### 返回值说明:

- BM\_JPU\_DEC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### bm\_jpu\_dec\_unload

该接口释放指定的解码设备节点。

#### 接口形式:

```
BmJpuDecReturnCodes bm_jpu_dec_unload(int device_index);
```

#### 参数说明:

- **device\_index**
- 解码设备 ID。

#### 返回值说明:

- BM\_JPU\_DEC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### bm\_jpu\_jpeg\_dec\_open

该接口打开一个解码器，根据传入的参数配置解码通道。

#### 接口形式:

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_open(BmJpuJPEGDecoder **jpeg_
↪ decoder,
                                     BmJpuDecOpenParams *open_params,
                                     unsigned int num_extra_framebuffers)
```

#### 参数说明:

- **jpeg\_decoder**  
指向解码器的二级指针，在接口内部完成初始化。
- **open\_params**  
打开解码器时的配置参数。
- **num\_extra\_framebuffers**  
需要额外申请的 framebuffer 个数。

#### 返回值说明:

- BM\_JPU\_DEC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### bm\_jpu\_jpeg\_dec\_close

该接口用来关闭解码器，释放资源。

#### 接口形式:

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_close(BmJpuJPEGDecoder *jpeg_
↪decoder);
```

#### 参数说明:

- jpeg\_decoder  
一个已经打开的解码器。

#### 返回值说明:

- BM\_JPU\_DEC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### bm\_jpu\_jpeg\_dec\_decode

该接口执行解码操作。

#### 接口形式:

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_decode(
    BmJpuJPEGDecoder *jpeg_decoder,
    uint8_t const *jpeg_data,
    size_t const jpeg_data_size);
```

#### 参数说明:

- jpeg\_decoder  
一个已经打开的解码器。
- jpeg\_data  
待解码的图像数据。
- jpeg\_data\_size  
待解码的图像数据大小。

#### 返回值说明:

- BM\_JPU\_DEC\_RETURN\_CODE\_OK: 成功
- 其他: 失败



### bm\_jpu\_jpeg\_dec\_get\_info

该接口从解码器获取解码信息。

#### 接口形式:

```
void bm_jpu_jpeg_dec_get_info(  
    BmJpuJPEGDecoder *jpeg_decoder,  
    BmJpuJPEGDecInfo *info);
```

#### 参数说明:

- **jpeg\_decoder**  
一个已经打开的解码器。
- **info**  
用于存储解码信息的数据结构。

#### 返回值说明:

- 无

### bm\_jpu\_jpeg\_dec\_frame\_finished

该接口释放一帧解码完成的 framebuffer。

#### 接口形式:

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_frame_finished(  
    BmJpuJPEGDecoder *jpeg_decoder,  
    BmJpuFramebuffer *framebuffer);
```

#### 参数说明:

- **jpeg\_decoder**  
一个已经打开的解码器。
- **framebuffer**  
一帧解码完成的 framebuffer。

#### 返回值说明:

- **BM\_JPU\_DEC\_RETURN\_CODE\_OK**: 成功
- 其他: 失败

### bm\_jpu\_jpeg\_dec\_flush

该接口用来刷新解码器，释放所有解码完成的 framebuffer。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_flush(BmJpuJPEGDecoder *jpeg_  
↪decoder);
```

参数说明：

- jpeg\_decoder  
一个已经打开的解码器。

返回值说明：

- BM\_JPU\_DEC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### bm\_jpu\_enc\_load

该接口根据传入的 ID 打开指定的编码设备节点，可以通过 bmlib 管理内存分配。

接口形式：

```
BmJpuEncReturnCodes bm_jpu_enc_load(int device_index);
```

参数说明：

- device\_index  
编码设备 ID。

返回值说明：

- BM\_JPU\_ENC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### bm\_jpu\_enc\_unload

该接口释放指定的编码设备节点。

接口形式：

```
BmJpuEncReturnCodes bm_jpu_enc_unload(int device_index);
```

参数说明：

- device\_index  
编码设备 ID。

**返回值说明:**

- BM\_JPU\_ENC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

**bm\_jpu\_jpeg\_enc\_open**

该接口打开一个编码器，并申请指定大小的 bitstream buffer。

**接口形式:**

```
BmJpuEncReturnCodes bm_jpu_jpeg_enc_open(
    BmJpuJPEGEncoder **jpeg_encoder,
    int bs_buffer_size,
    int device_index
);
```

**参数说明:**

- **jpeg\_encoder**  
指向编码器的二级指针，在接口内部完成初始化。
- **bs\_buffer\_size**  
bistream buffer 的大小，输入为 0 则默认申请 5MB。
- **device\_index**  
编码设备 ID。

**返回值说明:**

- BM\_JPU\_ENC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

**bm\_jpu\_jpeg\_enc\_close**

该接口用来关闭编码器，释放资源。

**接口形式:**

```
BmJpuEncReturnCodes bm_jpu_jpeg_enc_close(BmJpuJPEGEncoder *jpeg_
↪encoder);
```

**参数说明:**

- **jpeg\_encoder**  
一个已经打开的编码器

**返回值说明:**

- BM\_JPU\_ENC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### **bm\_jpu\_jpeg\_enc\_encode**

该接口执行编码操作。

#### **接口形式:**

```
BmJpuEncReturnCodes bm_jpu_jpeg_enc_encode(  
    BmJpuJPEGEncoder *jpeg_encoder,  
    BmJpuFramebuffer const *framebuffer,  
    BmJpuJPEGEncParams const *params,  
    void **acquired_handle,  
    size_t *output_buffer_size  
);
```

#### **参数说明:**

- **jpeg\_encoder**  
一个已经打开的编码器。
- **framebuffer**  
输入的 frame 数据。
- **params**  
编码相关参数。
- **acquired\_handle**  
用于存放编码数据的位置，由用户指定。如果为 NULL，则通过 write\_output\_data 接口输出。
- **output\_buffer\_size**  
输出数据 buffer 的大小，单位：byte。

#### **返回值说明:**

- BM\_JPU\_ENC\_RETURN\_CODE\_OK: 成功
- 其他: 失败

### acquire\_output\_buffer

该接口用于获取 buffer 接收编码数据。

接口形式:

```
void* acquire_output_buffer(void *context, unsigned int size, void **acquired_
↪handle);
```

参数说明:

- **context**  
输出 buffer 上下文。
- **size**  
输出 buffer 大小, 单位: byte。
- **acquired\_handle**  
输出 buffer 的起始地址。

### finish\_output\_buffer

该接口用于释放上述接口获取的 buffer。

接口形式:

```
void finish_output_buffer(void *context, void *acquired_handle);
```

参数说明:

- **context**  
输出 buffer 上下文。
- **acquired\_handle**  
输出 buffer 的起始地址。

## 2.5.4 JPEG 测试用例说明

### bmjpegdec

代码请参考 example/jpeg\_dec\_test.c

参数说明

```
usage: bmjpegdec [option]
option:
    -i input_file
```

(续下页)

(接上页)

```

-o output_file
-n loop_num
-c crop function(0:disable 1:enable crop)
-g rotate (default 0) [rotate mode[1:0] 0:No rotate 1:90 2:180 3:270] [rotator F]
↪ mode[2]:vertical flip] [rotator mode[3]:horizontal flip]
-s scale (default 0) -> 0 to 3
-r roi_x,roi_y,roi_w,roi_h

```

### 单路解码

```

bmjpegdec -i JPEG_1920x1088_yuv420_planar.jpg -o out_1920x1088_yuv420_
↪ planar.yuv -n 1

```

### 单路循环解码

```

bmjpegdec -i JPEG_1920x1088_yuv420_planar.jpg -o out_1920x1088_yuv420_
↪ planar.yuv -n 10

```

## bmjpegenc

代码请参考 example/jpeg\_enc\_test.c

### 参数说明

```

usage: bmjpegenc [option]
option:
  -f pixel format: 0.YUV420(default); 1.YUV422; 2.YUV444; 3.YUV400. (optional)
  -w actual width
  -h actual height
  -y luma stride (optional)
  -c chroma stride (optional)
  -v aligned height (optional)
  -i input file
  -o output file
  -n loop num
  -g rotate (default 0) [rotate mode[1:0] 0:No rotate 1:90 2:180 3:270] [rotator F]
↪ mode[2]:vertical flip] [rotator mode[3]:horizontal flip]

```

### 单路编码

```

bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪ 1920x1088_yuv420_planar.jpg -n 1

```

### 单路循环编码

```

bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪ 1920x1088_yuv420_planar.jpg -n 10000

```

### 旋转

```
bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪1920x1088_yuv420_planar.jpg -n 1 -g 1
```

## 镜像

```
bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪1920x1088_yuv420_planar.jpg -n 1 -g 4
```

## bmjpegmulti

代码请参考 example/jpeg\_multi\_test.c

### 32 路解码

首先编写配置文件 **multi**.  
 ↪**lst**, 内容如下: (第一行表示路数和每路的循环次数, 之后每一行表示每一路的配置)

```
32 100
JPEG_352x288_yuv420_planar.jpg 32 1 0 0
JPEG_352x288_yuv420_planar.jpg 32 1 0 0
...(重复到32行)
```

然后执行 **bmjpegmulti -f multi.lst**  
 选项输入 **30**

### 32 路编码

首先编写配置文件 **enc.cfg**, 内容如下:

```
YUV_SRC_IMG JPEG_352x288_yuv420_planar.yuv
FRAME_FORMAT 0
PICTURE_WIDTH 352
PICTURE_HEIGHT 288
IMG_FORMAT 0
```

然后编写配置文件 **multi**.  
 ↪**lst**, 内容如下: (第一行表示路数和每路的循环次数, 之后每一行表示每一路的配置)

```
32 1000000
enc.cfg 32 0 0 0
enc.cfg 32 0 0 0
...(重复到32行)
```

然后执行 **bmjpegmulti -f multi.lst**  
 选项输入 **30**

## 单路 YUV400 编码

```
bmjpegmulti -t 1 -i JPEG_1920x1088_yuv400_planar.yuv -o OUT400.jpg -w 1920 -h 1088 -s 4 -f 0 -n 4
```

### 单路 YUV420P 编码

```
bmjpegmulti -t 1 -i JPEG_1920x1088_yuv420_planar.yuv -o OUT420_planar.jpg -w 1920 -h 1088 -s 0 -f 0 -n 4
```

### 单路 YUV422P 编码

```
bmjpegmulti -t 1 -i JPEG_1920x1088_yuv422_planar.yuv -o OUT422_planar.jpg -w 1920 -h 1088 -s 1 -f 0 -n 4
```

## 2.6 SOPHGO Video Decoder 使用指南

### 2.6.1 简介

#### 概述

VDEC 模块提供驱动视频解码硬件工作的对应接口，实现视频解码功能。

BM1688 VDEC 模块支持 H.264 和 H.265 解码，支持对 16 路 1080P 视频同时以 30fps 的性能进行解码。

#### 定义及缩写

| 缩写        | 含义     |
|-----------|--------|
| VPU       | 视频处理单元 |
| VDEC      | 视频解码器  |
| core      | 核心     |
| BitStream | 输入码流数据 |
| Frame     | 帧      |
| Buffer    | 缓冲区    |
| channel   | 通道     |

### 2.6.2 VDEC 数据类型介绍

#### BMVidDecRetStatus

定义了解码器错误返回值类型。

#### 枚举定义

```
typedef enum
{
    BM_ERR_VDEC_INVALID_CHNID = -27,
```

(续下页)



(接上页)

```

BM_ERR_VDEC_ILLEGAL_PARAM,
BM_ERR_VDEC_EXIST,
BM_ERR_VDEC_UNEXIST,
BM_ERR_VDEC_NULL_PTR,
BM_ERR_VDEC_NOT_CONFIG,
BM_ERR_VDEC_NOT_SUPPORT,
BM_ERR_VDEC_NOT_PERM,
BM_ERR_VDEC_INVALID_PIPEID,
BM_ERR_VDEC_INVALID_GRPID,
BM_ERR_VDEC_NOMEM,
BM_ERR_VDEC_NOBUF,
BM_ERR_VDEC_BUF_EMPTY,
BM_ERR_VDEC_BUF_FULL,
BM_ERR_VDEC_SYS_NOTREADY,
BM_ERR_VDEC_BADADDR,
BM_ERR_VDEC_BUSY,
BM_ERR_VDEC_SIZE_NOT_ENOUGH,
BM_ERR_VDEC_INVALID_VB,
BM_ERR_VDEC_ERR_INIT,
BM_ERR_VDEC_ERR_INVALID_RET,
BM_ERR_VDEC_ERR_SEQ_OPER,
BM_ERR_VDEC_ERR_VDEC_MUTEX,
BM_ERR_VDEC_ERR_SEND_FAILED,
BM_ERR_VDEC_ERR_GET_FAILED,
BM_ERR_VDEC_BUTT,
BM_ERR_VDEC_FAILURE
}BMVidDecRetStatus;

```

### 参数含义

- **BM\_ERR\_VDEC\_INVALID\_CHNID**: 无效的 channel id。
- **BM\_ERR\_VDEC\_NULL\_PTR**: 空指针。
- **BM\_ERR\_VDEC\_NOBUF**: 无效的缓冲区。
- **BM\_ERR\_VDEC\_BUF\_FULL**: 缓冲区空。
- **BM\_ERR\_VDEC\_BUF\_FULL**: 缓冲区满。
- **BM\_ERR\_VDEC\_BUSY**: 解码器忙。
- **BM\_ERR\_VDEC\_FAILURE**: 解码器一般错误。

## BMDecStatus

用于指示解码器的状态。

### 枚举定义

```
typedef enum {
    BMDEC_UNCREATE,
    BMDEC_UNLOADING,
    BMDEC_UNINIT,
    BMDEC_INITING,
    BMDEC_DECODING,
    BMDEC_FRAME_BUF_FULL,
    BMDEC_ENDOF,
    BMDEC_STOP,
    BMDEC_HUNG,
    BMDEC_CLOSE,
    BMDEC_CLOSED,
} BMDecStatus;
```

### 参数含义

目前只有以下几个状态有效：

- **BMDEC\_UNINIT**: 解码器未进行初始化
- **BMDEC\_INITING**: 解码器正在进行初始化
- **BMDEC\_DECODING**: 解码器正在解码
- **BMDEC\_STOP**: 解码器解码结束

## BMVidDecParam

BMVidDecParam 用于设置解码器的初始化参数，在调用接口 `bmvpvpu_dec_create` 前需要创建 BMVidDecParam 对象，并对其进行初始化。

### 结构体定义

```
typedef struct {
    int    streamFormat;
    int    wtlFormat;
    int    enableCrop;
    int    cbcrInterleave;
    int    nv21;
    int    secondaryAXI;
    int    streamBufferSize;
    int    mp4class;
    int    bsMode;
    int    extraFrameBufferNum;
    int    frameDelay;
    int    pcie_board_id;
```

(续下页)

(接上页)

```

int    pcie_no_copyback;
int    enable_cache;
int    skip_mode;
int    perf;
int    core_idx;
int    reserved[13];
} BMVidDecParam;

```

### 参数含义

- **streamFormat:**  
设置输入码流类型，0 为 H.264(AVC)，12 为 H.265(HEVC)。
- **wtlFormat:**  
设置输出数据类型。  
设置为 0，则根据 yuv 类型输出对应的 yuv 数据；  
设置为 101，则输出压缩的 fbc 数据。
- **cbrInterleave:**  
设置 yuv 格式。  
设置为 0，Cb、Cr 数据分别存储在不同的通道中；  
设置为 1，Cb、Cr 数据存储在同一通道中。
- **nv21:**  
仅当 cbrInterleave 设置为 1 时有效。  
设置为 0，以 NV12 格式输出；  
设置为 1，以 NV21 格式输出。
- **secondaryAXI:**  
BM1688 中不在需要设置此参数，  
SDK 中会根据码流类型，自动选择是否开启 secondary AXI 功能。
- **streamBufferSize:**  
设置输入码流的缓冲区大小。  
若设置为 0，则默认缓冲区大小为 0x700000。
- **bsMode:**  
设置解码器工作方式。  
0 以 INTERRUPT 模式工作；2 以 PIC\_END 模式工作。
- **extraFrameBufferNum:**  
设置 Frame Buffer 的数量。

## BMVidStream

通过 BMVidStream 对象，向解码器传递码流数据。

### 结构体定义

```
typedef struct BMVidStream {
    unsigned char* buf;
    unsigned int length;
    unsigned char* header_buf;
    unsigned int header_size;
    unsigned char* extradata;
    unsigned int extradata_size;
    unsigned char end_of_stream;
    u64 pts;
    u64 dts;
} BMVidStream;
```

### 参数含义

- **buf、length:**  
BitStream Buffer 的地址和大小。
- **end\_of\_stream:**  
码流结束标志。当码流读取完成后，需要将这个标志位置 1。
- **pts、dts:**  
时间戳
- BM1688 中，不再接受 header 和 extradata 的数据。为了和前序产品保持一致，以上变量仍然存在。

## BMVidFrame

BMVidFrame 用于接收解码器输出的帧信息。

### 结构体定义

```
typedef struct BMVidFrame {
    int picType;
    unsigned char* buf[8];
    int stride[8];
    unsigned int width;
    unsigned int height;
    int frameFormat;
    int interlacedFrame;
    int lumaBitDepth;
    int chromaBitDepth;
    int cbrInterleave;
    int nv21;
    int endian;
    int sequenceNo;
```

(续下页)

(接上页)

```

int      frameIdx;
u64      pts;
u64      dts;
int      colorPrimaries;
int      colorTransferCharacteristic;
int      colorSpace;
int      colorRange;
int      chromaLocation;
int      size;
unsigned int  coded_width;
unsigned int  coded_height;
int      compressed_mode;
} BMVidFrame;

```

### 参数含义

- **picType:**

表示当前 Frame 的类型，对应关系如下：

表 2.2: picType 对应关系

| picType | 类型          |
|---------|-------------|
| 0       | I picture   |
| 1       | P picture   |
| 2       | B picture   |
| 4       | IDR picture |

- **buf:**

存放输出数据的地址。各通道对应的含义如下表：

表 2.3: buf 对应关系

| channel | YUV420P   | NV12/NV21   | FBC             |
|---------|-----------|-------------|-----------------|
| 0       | Y 分量虚拟地址  | Y 分量虚拟地址    | /               |
| 1       | Cb 分量虚拟地址 | CbCr 分量虚拟地址 | /               |
| 2       | Cr 分量虚拟地址 | /           | /               |
| 3       | /         | /           | /               |
| 4       | Y 分量物理地址  | Y 分量物理地址    | Y 分量物理地址        |
| 5       | Cb 分量物理地址 | CbCr 分量物理地址 | Cb 分量物理地址       |
| 6       | Cr 分量物理地址 | /           | Y table 分量物理地址  |
| 7       | /         | /           | Cb table 分量物理地址 |

- **stride:**

和 buf 对应，存放对应通道的宽度，该宽度是进行对齐后的宽度。

对于 FBC 数据，stride 存放的数据稍有不同。

channel 0 和 4，存放 Y 分量的宽度；

channel 1 和 5, 存放 Cb 分量的宽度;

channel 2 和 6, 存放 Y table 的长度;

channel 3 和 7, 存放 Cb table 的长度。

- **width:**  
存放 Frame 的宽度。
- **height:**  
存放 Frame 的高度。
- **frameFormat:**  
存放图像的格式。
- **interlacedFrame:**  
图像扫描方式。
- **lumaBitDepth:**  
亮度数据的深度。
- **chromaBitDepth:**  
色度数据的深度。
- **cbcrInterleave:**  
表示色度分量的存储方式。  
cbcrInterleave 为 0, Cb 和 Cr 分量存储在不同的内存空间中;  
cbcrInterleave 为 1, Cb 和 Cr 分量存储在同一个内存空间中。
- **nv21:**  
表示 YUV 数据的存储格式, 仅当 cbcrInterleave=1 时有效。  
nv21=0, 以 Nv12 格式存储;  
nv21=1, 以 NV21 格式存储。
- **endian:**  
表示帧缓冲区的段序。  
endian=0, 以小端模式存储;  
endian=1, 以大端模式存储;  
endian=2, 以 32 位小端模式存储;  
endian=3, 以 32 位大端模式存储。
- **sequenceNo:**  
表示码流序列的状态。当码流序列改变时, sequenceNo 的值会进行累加。
- **frameIdx:**  
图像帧缓冲区的索引。用于表示该帧缓冲区在解码器中位置。
- **pts:**  
显示时间戳。

- **dts:**  
解码时间戳。
- **size:**  
帧缓冲区的大小。
- **coded\_width:**  
用于显示的图片宽度。
- **coded\_height:**  
用于显示的图片高度。
- **compressed\_mode:**  
表示解码器输出数据的格式。

## CropRect

CropRect 用于保存图像的剪裁信息。

### 结构体定义

```
typedef struct {
    unsigned int left;
    unsigned int top;
    unsigned int right;
    unsigned int bottom;
} CropRect;
```

### 参数含义

- **left:**  
剪裁框左上角相对于像素原点的水平偏移量。
- **top:**  
剪裁框左上角相对于像素原点的垂直偏移量。
- **right:**  
剪裁框右下角相对于像素原点的水平偏移量。
- **bottom:**  
剪裁框右下角相对于像素原点的垂直偏移量。

## BMVidStreamInfo

BMVidStreamInfo 用于接收输入码流的信息。

### 结构体定义

```
typedef struct BMVidStreamInfo {
    int    picWidth;
    int    picHeight;
```

(续下页)

(接上页)

```

int      fRateNumerator;
int      fRateDenominator;
CropRect picCropRect;
int      mp4DataPartitionEnable;
int      mp4ReversibleVlcEnable;
int      mp4ShortVideoHeader;
int      h263AnnexJEnable;
int      minFrameBufferCount;
int      frameBufDelay;
int      normalSliceSize;
int      worstSliceSize;
int      maxSubLayers;
int      profile;
int      level;
int      tier;
int      interlace;
int      constraint_set_flag[4];
int      direct8x8Flag;
int      vc1Psf;
int      isExtSAR;
int      maxNumRefFrmFlag;
int      maxNumRefFrm;
int      aspectRateInfo;
int      bitRate;
int      mp2LowDelay;
int      mp2DispVerSize;
int      mp2DispHorSize;
unsigned int userDataHeader;
int      userDataNum;
int      userDataSize;
int      userDataBufFull;
int      chromaFormatIDC;
int      lumaBitdepth;
int      chromaBitdepth;
int      seqInitErrReason;
int      warnInfo;
unsigned int sequenceNo;
} BMVidStreamInfo;

```

### 参数含义

- **picWidth:**  
图片宽度。
- **picHeight:**  
图片高度。
- **fRateNumerator:**  
帧率分数的分子。
- **fRateDenominator:**  
帧率分数的分母。



- **picCropRect:**  
剪裁信息，详细信息可以参考 CropRect。
- **minFrameBufferCount:**  
解码器所需要的帧缓冲区的最小数量。
- **frameBufDelay:**  
用于缓冲解码图像重排序的最大显示帧缓冲延迟。
- **profile:**  
H.264/H.265 的配置文件索引。
- **level:**  
H.264/H.265 的级别索引。
- **interlace:**  
interlace=1 时，码流被解码为逐行帧或隔行帧；否则解码为逐行帧。
- **bitRate:**  
BitStream 写入时的比特率。
- **lumaBitdepth:**  
亮度数据的深度。
- **chromaBitdepth:**  
色度数据的深度。
- **sequenceNo:**  
表示码流序列的状态。当码流序列改变时，sequenceNo 的值会进行累加。

### 2.6.3 VDEC API 介绍

#### bmvpvpu\_dec\_create

bmvpvpu\_dec\_create 用来创建一个解码器通道。

#### 接口形式

```
int bmvpvpu_dec_create (
    BMVidCodHandle* pVidCodHandle,
    BMVidDecParam decParam );
```

#### 参数说明

- **pVidCodHandle:**  
输出参数。解码器句柄，当解码器创建成功后将会返回一个句柄，用句柄可以对解码器进行后续的操作。
- **decParam:**  
输入参数。解码器初始化参数。

#### 返回值

当解码器创建成功，会返回 0，否则将返回对应的错误码。

### bmvpvu\_dec\_decode

利用 `bmvpvu_dec_decode` 将 BitStream 送入 VDEC 进行解码。BM1688 支持两种解码模式，分别是 INTERRUPT 和 PIC\_END 模式。

**INTERRUPT 模式**是按照预先设置的 BitStream Buffer 大小，每次送入固定大小的码流数据，并不存在帧的概念。

**PIC\_END 模式**是根据 H.264/H.265 协议，预先解析出一帧数据的位置，每次只向解码器传输一帧数据对应的 BitStream，因此在创建解码器时，需要合理考虑 `streamBufferSize` 的值。

#### 接口形式

```
int bmvpvu_dec_decode (
    BMVidCodHandle vidCodHandle,
    BMVidStream vidStream );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **vidStream:**  
输入参数。BitStream 的地址和大小。

#### 返回值

当码流成功送入解码器，会返回 0，否则将返回对应的错误码。返回错误并不代表解码器工作异常。若返回 `BM_ERR_VDEC_BUF_FULL`，需要检查解码器状态，并再次尝试送 BitStream。

### bmvpvu\_dec\_get\_output

`bmvpvu_dec_get_output` 这个接口用于来获取解码器的输出数据。

#### 接口形式

```
int bmvpvu_dec_get_output (
    BMVidCodHandle vidCodHandle,
    BMVidFrame *bmFrame );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **bmFrame:**  
输出参数。用于接收解码器的输出数据。

#### 返回值

当成功获得解码数据，会返回 0，否则返回对应的错误码。返回错误码并不代表解码器工作异常，以下情况都有可能造成返回错误码：

1. 没有输入 BitStream 数据；
2. 解码尚未完成，输出数据未准备好；
3. 解码器关闭；
4. 解码器异常。

### **bmvpv\_dec\_clear\_output**

当 Frame Buffer 不再使用时，需要调用 `bmvpv_dec_clear_output` 对其进行释放。否则解码器可能因为没有足够的 Frame Buffer 而工作异常。

#### **接口形式**

```
int bmvpv_dec_clear_output (
    BMVidCodHandle vidCodHandle,
    BMVidFrame *frame );
```

#### **参数说明**

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **frame:**  
输入参数。需要释放的 `bmFrame` 对象的地址。

#### **返回值**

当 `frame` 释放成功，会返回 0，否则返回对应的错误码。

### **bmvpv\_dec\_flush**

当 BitStream 全部送入解码器后，并不代表解码已经完成，还需要等待解码器将全部的 Frame 输出。用 `bmvpv_dec_flush` 来刷出剩余的 Frame。

#### **接口形式**

```
int bmvpv_dec_flush (
    BMVidCodHandle vidCodHandle );
```

#### **参数说明**

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

#### **返回值**

当 Frame 全部取出后，将会返回 0。

### bmvpv\_dec\_get\_all\_frame\_in\_buffer

同样用于在码流传送完毕后，获取解码器中的剩余的 Frame 数据。和 bmvpv\_dec\_flush 不同之处在于，bmvpv\_dec\_get\_all\_frame\_in\_buffer 不会阻塞。

#### 接口形式

```
int bmvpv_dec_get_all_frame_in_buffer (
    BMVidCodHandle vidCodHandle );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

#### 返回值

当 Frame 全部拿出解码器后，将会返回 0。

### bmvpv\_dec\_delete

当解码任务完成后，调用 bmvpv\_dec\_delete 销毁解码器，释放解码器占用的资源

#### 接口形式

```
int bmvpv_dec_delete (
    BMVidCodHandle vidCodHandle );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

#### 返回值

解码器销毁成功后，会返回 0，否则返回对应的错误码。

### bmvpv\_dec\_get\_caps

bmvpv\_dec\_get\_caps 用于获取解码器信息，主要是送入解码器的码流信息。能够获取到的信息可以参考 BMVidStreamInfo 结构体的定义。

#### 接口形式

```
int bmvpv_dec_get_caps (
    BMVidCodHandle vidCodHandle,
    BMVidStreamInfo *streamInfo );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

- **streamInfo:**

输出参数。接收解码器信息。

### 返回值

成功获取解码器信息，返回 0，否则返回对应的错误码。

### `bmvpvpu_dec_get_status`

`bmvpvpu_dec_get_status` 用于获取解码器的工作状态。目前所支持的解码器状态可以参考 `BMDecStatus` 的定义。

### 接口形式

```
BMDecStatus bmvpvpu_dec_get_status (  
    BMVidCodHandle vidCodHandle );
```

### 参数说明

- **vidCodHandle:**

输入参数。解码器句柄，在使用该接口前需要先创建解码器。

### 返回值

若成功获取到解码器状态，将会返回对应的状态，反则将返回错误码。

### `bmvpvpu_dec_get_all_empty_input_buf_cnt`

用于获取空闲的 BitStream Buffer 的数量。

### 接口形式

```
int bmvpvpu_dec_get_all_empty_input_buf_cnt (  
    BMVidCodHandle vidCodHandle );
```

### 参数说明

- **vidCodHandle:**

输入参数。解码器句柄，在使用该接口前需要先创建解码器。

### 返回值

若成功获取到 BitStream 的数量，则返回空闲 Buffer 的数量，否则返回错误码。

### bmvpv\_dec\_get\_pkt\_in\_buf\_cnt

用于获取被占用的 BitStream Buffer 的数量。

#### 接口形式

```
int bmvpv_dec_get_pkt_in_buf_cnt (
    BMVidCodHandle vidCodHandle );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

#### 返回值

若成功获取到 BitStream 的数量，则返回被占用 Buffer 的数量，否则返回错误码。

### bmvpv\_dec\_dump\_stream

用于保存输入码流数据，目前仅支持保存 INTERRUPT 模式的数据；若为 PIC\_END 模式，保存的数据可能出现无法播放的情况（缺少一些非显示帧的信息）。该接口已经写入 bmvpv\_dec\_decode 中，通过设置环境变量 BMVPV\_DEC\_DUMP\_NUM 可以使能该功能。

#### 接口形式

```
int bmvpv_dec_dump_stream (
    BMVidCodHandle vidCodHandle,
    BMVidStream vidStream );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **vidStream:**  
输入参数。BitStream 的地址和大小。

#### 返回值

码流保存成功，则返回 0，否则返回错误码。

### `bmvpvu_dec_get_core_idx`

用于获取 VPU 的 core id。对于 BM1688，有两个解码 core，分别是 core 0 和 core 1。

#### 接口形式

```
int bmvpvu_dec_get_core_idx (
    BMVidCodHandle handle );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

#### 返回值

成功则返回 VPU 的 core id，否则返回错误码。

### `bmvpvu_dec_get_inst_idx`

用于获取 VDEC 的 channel id。对于 BM1688 来说，每个 core 最多可以申请 32 个 channel。

#### 接口形式

```
int bmvpvu_dec_get_inst_idx (
    BMVidCodHandle vidCodHandle );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

#### 返回值

成功则返回 VDEC 的 channel id，否则返回错误码。

### `bmvpvu_dec_get_device_fd`

用于获取 VDEC channel 对应的设备节点 id。

#### 接口形式

```
int bmvpvu_dec_get_device_fd (
    BMVidCodHandle vidCodHandle );
```

#### 参数说明

- **vidCodHandle:**  
输入参数。解码器句柄，在使用该接口前需要先创建解码器。

#### 返回值

成功则返回设备节点的 id，否则返回错误码。

### 2.6.4 VDEC API 测试例程

针对 VDEC 提供了专用的测试例程 `bm_test`。其中调用了 VDEC API 来实现对本地 H.264/H.265 码流文件进行解码。

`bm_test` 提供了以下可配置参数

- **-h** 查看提示信息，可以看到 `bm_test` 的可选参数以及对应的参数说明。
- **-v** 设置 LOG 等级。0: none; 1: error; 2: warning; 3: info; 4: trace。缺省值为 1。
- **-c** 设置输出校验方式。0: 不进行输出校验; 1: 直接对比 yuv 数据; 2: 对比输出 yuv 的 md5 值。  
开启输出校验前需提前准备好参考文件，通过 `-comp-skip` 设置对比的频率。缺省值为 0。
- **-n** 设置解码的轮数。缺省值为 1。
- **-m** 设置解码模式。0: 设置为 INTERRUPT 模式; 2: 设置为 PIC\_END 模式。
- **-input** 设置输入文件。
- **-output\_format** 设置解码输出的类型。0: 输出 yuv420p; 1: 输出 nv12; 2: 输出 nv21。缺省值为 0。
- **-stream-type** 设置输入码流类型。0: H.264; 12: H.265。
- **-ref-yuv** 设置参考文件路径。当开启输出校验后，必须设置此参数，否则不会进行输出校验。
- **-instance** 设置解码线程数。
- **-write\_yuv** 设置输出帧数。这个参数只用于设置需要写文件的帧数，并不是解码帧数。
- **-wtl-format** 设置输出模式。0: 输出 YUV; 1: 输出 FBC。
- **-read-block-len** 设置输入缓冲区大小。缺省值为 0x80000。

#### 示例 1

采用 INTERRUPT 模式对 H.264 码流进行解码，采用 yuv420p 格式进行输出。

```
bm_test -c 0 -n 1 -m 0 --input wkc_h264_100.264 --output_format 0 --instance 1
```

#### 示例 2

采用 PIC\_END 模式对 H.265 码流进行解码，采用 yuv420p 格式进行输出。

```
bm_test -c 0 -n 1 -m 2 --stream_type 12 --input wkc_h264_100.265 --output_format 0 --  
↪instance 1
```



### 示例 3

采用 INTERRUPT 模式对 H.264 码流进行解码，采用 nv12 格式进行输出，并且开启 4 个解码线程。

```
bm_test -c 0 -n 1 -m 0 --input wkc_h264_100.264 --output_format 1 --instance 4
```

### 示例 4

采用 INTERRUPT 模式对 H.264 码流进行解码，采用 nv12 格式进行输出，并且开启 yuv 对比。

```
bm_test -c 1 -n 1 -m 0 --input wkc_h264_100.264 --output_format 1 --ref-yuv wkc_h264_100.
↪yuv --instance 1
```

### 示例 5

采用 PIC\_END 模式对 H.265 码流进行解码，采用 nv12 格式进行输出，并且开启 md5 对比，将前 10 帧写入文件。

```
bm_test -c 2 -n 1 -m 2 --input wkc_h264_100.265 --output_format 1 --ref-yuv wkc_h264_100.
↪md5 --instance 1 --write_yuv 10
```

## 2.7 SOPHGO Video Encoder 使用指南

### 2.7.1 简介

Encoder 模块主要包括了视频的编码，该模块囊括了所以对用户开放的接口以及其参数详解

### 2.7.2 数据结构说明

#### 1. BmVpuEncInitialInfo

```
typedef struct
{
    uint32_t min_num_rec_fb;
    uint32_t min_num_src_fb;
    uint32_t framebuffer_alignment;
    BmVpuFbInfo src_fb;
} BmVpuEncInitialInfo;
```

#### 参数说明

- min\_num\_src\_fb

输入的 yuv 最小数量，用户分配到数量小于此值会影像编码

- src\_fb

输入 yuv 的宽高等信息

## 2. BmVpuFbInfo

### 参数说明

- width

原始视频的宽

- height

原始视频的高

- y\_stride

对齐后的 Y 通道宽

- c\_stride

对齐后的 UV 通道宽

- h\_stride

对齐后的高

- y\_size

对齐后的 Y 通道大小

- c\_size

对齐后的 UV 通道大小

- size

对齐后的 YUV 大小

## 3. BmVpuCodecFormat

```
typedef enum
{
    BM_VPU_CODEC_FORMAT_H264,
    BM_VPU_CODEC_FORMAT_H265,
} BmVpuCodecFormat;
```

### 参数说明

- 编码器支持的编码格式

BM\_VPU\_CODEC\_FORMAT\_H264 编码类型是 h264

BM\_VPU\_CODEC\_FORMAT\_H265 编码类型是 h265

#### 4. BmVpuColorFormat

```
typedef enum
{
    BM_VPU_COLOR_FORMAT_YUV420 = 0,
    BM_VPU_COLOR_FORMAT_NV12  = 1,
    BM_VPU_COLOR_FORMAT_NV21  = 2,
    BM_VPU_COLOR_FORMAT_YUV422 = 3,
    BM_VPU_COLOR_FORMAT_NV16  = 4,
    BM_VPU_COLOR_FORMAT_YUV444 = 5,
    BM_VPU_COLOR_FORMAT_YUV400 = 6
} BmVpuColorFormat;
```

#### 参数说明

- 编码器输入 yuv 格式

目前支持 nv12, nv21, yuv420p

#### 5. BmVpuEncOpenParams

```
typedef struct
{
    BmVpuCodecFormat codec_format;
    BmVpuColorFormat color_format;
    uint32_t frame_width;
    uint32_t frame_height;
    uint32_t timebase_num;
    uint32_t timebase_den;
    uint32_t fps_num;
    uint32_t fps_den;
    int64_t bitrate;
    uint64_t vbv_buffer_size;
    int cqp;
    int enc_mode;
    int max_num_merge;
    int enable_constrained_intra_prediction;
    int enable_wp;
    int disable_deblocking;
    int offset_tc;
    int offset_beta;
    int enable_cross_slice_boundary;
    int enable_nr;
    union
```

(续下页)

(接上页)

```

{
    BmVpuEncH264Params h264_params;
    BmVpuEncH265Params h265_params;
};
int chroma_interleave;
int soc_idx;
int gop_preset;
int intra_period;
int bg_detection;
int mb_rc;
int delta_qp;

/* minimum QP for rate control
 * Default value is 8 */
int min_qp;
/* maximum QP for rate control
 * Default value is 51 */
int max_qp;

/* roi encoding flag
 * Default value is 0 */
int roi_enable;
/* set cmd queue depth
 * Default value is 4
 * the value must 1 <= value <= 4*/
int cmd_queue_depth;
} BmVpuEncOpenParams;

```

### 参数说明

- BmVpuCodecFormat codec\_format  
设置编码类型
- BmVpuColorFormat color\_format  
设置编码输入的 yuv 格式
- uint32\_t frame\_width  
设置编码视频的宽
- uint32\_t frame\_height  
设置编码视频的高
- uint32\_t timebase\_num  
设置帧率 num
- uint32\_t timebase\_den  
设置帧率 den
- int64\_t bitrate

设置码率, 如果设置了 `cqp`, 则 `bitrate` 无效

- `int cqp`

设置 `qp`, 不与码率同时设置

- `int enc_mode`

0: Custom mode.

1: 编码速度慢, 画质最高.

2: 编码速度正常, 画质正常

3: 编码速度高, 画质低

- `int max_num_merge`

- `int gop_preset`

1: all I, all Intra, gopsize = 1

2: I-P-P, consecutive P, cyclic gopsize = 1

3: I-B-B-B, consecutive B, cyclic gopsize = 1

4: I-B-P-B-P, gopsize = 2

5: I-B-B-B-P, gopsize = 4

6: I-P-P-P-P, consecutive P, cyclic gopsize = 4

7: I-B-B-B-B, consecutive B, cyclic gopsize = 4

8: Random Access, I-B-B-B-B-B-B-B, cyclic gopsize = 8

- `int intra_period`

GOP size, 默认值是 60

- `int min_qp`

设置最小 `qp` 值

- `int max_qp`

设置最大 `qp` 值

- `int cmd_queue_depth`

编码队列深度, 可提升编码器性能, 同时需要消耗一定的物理内存

## 6. BmVpuEncodedFrame

```
typedef struct
{
    uint8_t *data;
    size_t data_size;
    size_t data_len;
    BmVpuFrameType frame_type;
    void *acquired_handle;
    void *context;
    uint64_t pts;
    uint64_t dts;
    int src_idx;
    bm_pa_t u64CustomMapPhyAddr;
    int avg_ctu_qp;
} BmVpuEncodedFrame;
```

### 参数说明

- uint8\_t \*data  
编码输出的码流数据
- size\_t data\_size  
编码输出的码流数据长度
- size\_t data\_len  
编码输出的码流数据占用内存大小
- BmVpuFrameType frame\_type  
帧类型，区分 I 帧 B 帧 P 帧
- uint64\_t pts  
解码时间戳
- uint64\_t dts  
显示时间戳
- int src\_idx  
用来标识输入的 YUV 索引，用户通过此索引释放 YUV
- bm\_pa\_t u64CustomMapPhyAddr  
使能 roi 时，roi 使用的物理内存
- int avg\_ctu\_qp  
ctu 平均 qp

## 7. BmVpuRawFrame

```
typedef struct
{
    BmVpuFramebuffer *framebuffer;
    void *context;
    uint64_t pts;
    uint64_t dts;
    int skip_frame;
    BmCustomMapOpt* customMapOpt;
} BmVpuRawFrame;
```

### 参数说明

- BmVpuFramebuffer \*framebuffer  
存储编码的 YUV 数据
- uint64\_t pts  
解码时间戳
- uint64\_t dts  
显示时间戳
- BmCustomMapOpt\* customMapOpt  
存放 roi 编码用的数据

## 8. BmVpuFramebuffer

```
typedef struct
{
    bm_device_mem_t *dma_buffer;
    int myIndex;
    unsigned int y_stride;
    unsigned int cbcr_stride;
    unsigned int width;
    unsigned int height;
    size_t y_offset;
    size_t cb_offset;
    size_t cr_offset;
    int already_marked;
    void *internal;
    void *context;
} BmVpuFramebuffer;
```

### 参数说明

- bm\_device\_mem\_t \*dma\_buffer  
保存 YUV 数据的物理内存

- int myIndex  
YUV 索引，用户设置，用于释放 YUV
- unsigned int y\_stride  
Y 通道对齐后的大小
- unsigned int cbr\_stride  
UV 通道对其后的大小
- unsigned int width  
编码 YUV 的宽
- unsigned int height  
编码 YUV 的高
- size\_t y\_offset  
Y 通道 offset
- size\_t cb\_offset  
U 通道 offset
- size\_t cr\_offset  
V 通道 offset

### 2.7.3 API 扩展说明

1. void bmvpu\_enc\_set\_default\_open\_params(BmVpuEncOpenParams \*open\_params, BmVpuCodecFormat codec\_format)

| 功能   | 设置编码器的默认参数                                                                                            |
|------|-------------------------------------------------------------------------------------------------------|
| 输入参数 | BmVpuEncOpenParams *open_params - 保存了编码器的参数<br>BmVpuCodecFormat codec_format - 编码器和解码器的选择,h264 和 h265 |
| 返回值  | 无                                                                                                     |
| 函数说明 | 在 bmvpu_enc_open 之前调用                                                                                 |



## 2. int bmvpu\_enc\_param\_parse(BmVpuEncOpenParams \*p, const char \*name, const char \*value)

| 功能   | 解析命令行参数, 存入 BmVpuEncOpenParams 中                                                      |
|------|---------------------------------------------------------------------------------------|
| 输入参数 | BmVpuEncOpenParams *p - 保存编码器的参数<br>const char *name - 选项名<br>const char *value - 选项值 |
| 返回值  | BM_SUCCESS - 执行成功 else - 执行失败                                                         |
| 函数说明 | 在 bmvpu_enc_open 之前调用                                                                 |

## 3. void bmvpu\_enc\_get\_bitstream\_buffer\_info(size\_t \*size, uint32\_t \*alignment)

| 功能   | 该函数得到编码器所需的 bitstream buffer 的大小 (size) 和所需要的 alignment 值                             |
|------|---------------------------------------------------------------------------------------|
| 输入参数 | BmVpuEncOpenParams *p - 保存编码器的参数<br>const char *name - 选项名<br>const char *value - 选项值 |
| 返回值  | 无                                                                                     |
| 函数说明 | 需要在 bmvpu_enc_open 之前调用                                                               |

## 4. int bmvpu\_enc\_open(BmVpuEncoder \*\*encoder, BmVpuEncOpenParams \*open\_params)

| 功能   | 打开一个新的编码器实例, 设置编码器参数并开始接收视频帧                                                                                         |
|------|----------------------------------------------------------------------------------------------------------------------|
| 输入参数 | BmVpuEncOpenParams *open_params - 编码器各项参数<br>BmVpuEncoder **encoder - 编码器实例, 接收编码器的属性和视频帧的部分设置, 例如设备 id、缓冲区设置和帧率、宽高等 |
| 返回值  | BM_SUCCESS - 打开成功 else - 打开失败                                                                                        |
| 函数说明 | 调用前需要确保 BmVpuEncOpenParams 中存储初始化编码器所需参数                                                                             |

## 5. int bmvpu\_enc\_close(BmVpuEncoder \*encoder)

| 功能   | 关闭编码器. 释放设备和通路                  |
|------|---------------------------------|
| 输入参数 | BmVpuEncoder *encoder - 待释放的编码器 |
| 返回值  | BM_SUCCESS - 关闭成功 else - 关闭失败   |
| 函数说明 | 无                               |

6. `int bmvpu_enc_get_initial_info(BmVpuEncoder *encoder, BmVpuEncInitialInfo *info)`

|      |                                                                                                                           |
|------|---------------------------------------------------------------------------------------------------------------------------|
| 功能   | 用来获取 encoder 的初始化信息, 可获取 src_buffer 和 rec_buffer 的最少所需数量, 并将相关信息填充到传入的结构体 BmVpuEncInitialInfo 中。                          |
| 输入参数 | BmVpuEncoder *encoder - 编码器<br>BmVpuEncInitialInfo *info - 这个结构体主要用于描述帧缓冲区在内存中的存储要求, 包括了帧的宽度、高度、各个分量的行跨度以及在 DMA 内存中的大小等信息 |
| 返回值  | BM_SUCCESS - 获取成功 else - 失败                                                                                               |
| 函数说明 | 需要在调用 bmvpu_enc_open 之后调用                                                                                                 |

7. `bm_handle_t bmvpu_enc_get_bmlib_handle(int soc_idx)`

|      |                                                                    |
|------|--------------------------------------------------------------------|
| 功能   | 该接口根据指定的设备索引值返回相应的管理物理内存的 bm_handle                                |
| 输入参数 | int soc_idx - 设备索引号                                                |
| 返回值  | bm_handle_t - 获取成功 NULL - 获取失败                                     |
| 函数说明 | 需要在 bmvpu_enc_load() 调用之后使用, 如果该 bm_handle 不存在, 则打印一条错误信息, 返回 NULL |

8. `int bmvpu_malloc_device_byte_heap(bm_handle_t bm_handle, bm_device_mem_t *pmem, unsigned int size, int heap_id_mask, int high_bit_first)`

|      |                                                                                                                                                                                                                                     |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 功能   | 分配设备内存以匹配指定的 heap_id_mask。                                                                                                                                                                                                          |
| 输入参数 | bm_handle_t bm_handle - 用于定位和操作特定的 bm 设备<br>bm_device_mem_t *pmem - 保存分配到的物理内存地址<br>unsigned int size - 需要分配的内存大小<br>int heap_id_mask - 代表了一个用于指定内存分配堆的掩模<br>int high_bit_first - 高位优先的标识, 决定了从掩模的高位开始查找还是低位开始。为 0 则低位优先, 1 则代表高位优先 |
| 返回值  | BM_SUCCESS - 分配成功 else - 分配失败                                                                                                                                                                                                       |
| 函数说明 | 如果 high_bit_first 为 1, 则将考虑 heap_id_mask 中的高位                                                                                                                                                                                       |

```
9.      int  bmvpu_fill_framebuffer_params(BmVpuFramebuffer *fb,  BmVpuFbInfo *info,
bm_device_mem_t *fb_dma_buffer, int fb_id, void *context)
```

|      |                                                                                                                                                                                                                                                                                               |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 功能   | 根据 bmvpu_fbinfo 获取的数据填充缓冲区结构体 BmVpuFramebuffer 结构体的字段, 同时也设置了指定的 DMA 缓冲区和上下文指针                                                                                                                                                                                                                |
| 输入参数 | BmVpuFramebuffer *fb - 一个实际的缓冲区, 通过接收 info 中的信息来设置缓冲区, 例如缓冲区索引、指针和帧的宽高以及 Y、U、V 三分量在帧缓冲区中的偏移等<br>BmVpuFbInfo *info - 存放缓冲区设置的相关信息<br>bm_device_mem_t *fb_dma_buffer - 指针指向了实际的 DMA 缓冲区的起始地址, 通过这个指针, 程序可以直接访问和操作 DMA 缓冲区中的数据, 而无需通过中央处理器进行复制或处理。<br>int fb_id - 缓冲区索引<br>void *context - 上下文信息 |
| 返回值  | BM_SUCCESS - 分配成功 else - 分配失败                                                                                                                                                                                                                                                                 |
| 函数说明 | 无                                                                                                                                                                                                                                                                                             |

```
10. int bmvpu_upload_data(int soc_idx, const uint8_t *host_va, int host_stride, uint64_t vpu_pa,
int vpu_stride, int width, int height);
```

|      |                                                                                                                                                                                                                  |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 功能   | 用于将系统内存中的数据写入指定的 soc 设备内存中                                                                                                                                                                                       |
| 输入参数 | int soc_idx - SoC 的索引值<br>const uint8_t *host_va - 数据的系统内存地址<br>int host_stride - 数据的在系统内存中的 stride<br>uint64_t vpu_pa - 数据要写入的物理内存地址<br>int vpu_stride - 数据的在设备内存中的 stride<br>int width - 帧宽<br>int height - 帧高 |
| 返回值  | BM_SUCCESS - 成功 else - 失败                                                                                                                                                                                        |
| 函数说明 | 仅支持 PCIe 模式                                                                                                                                                                                                      |

11. `int bmvpu_download_data(int soc_idx, const uint8_t *host_va, int host_stride, uint64_t vpu_pa, int vpu_stride, int width, int height);`

| 功能   | 用于从指定设备的内存中的数据中读取数据                                                                                                                                                                                                                                                                                          |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 输入参数 | <code>int soc_idx</code> - SoC 的索引值<br><code>const uint8_t *host_va</code> - 需要接收数据的系统内存地址<br><code>int host_stride</code> - 数据的在系统内存中的 stride<br><code>uint64_t vpu_pa</code> - 数据的物理内存地址<br><code>int vpu_stride</code> - 数据的在设备内存中的 stride<br><code>int width</code> - 帧宽<br><code>int height</code> - 帧高 |
| 返回值  | BM_SUCCESS - 成功 else - 失败                                                                                                                                                                                                                                                                                    |
| 函数说明 | 仅支持 PCIe 模式                                                                                                                                                                                                                                                                                                  |

12. `char const *bmvpu_color_format_string(BmVpuColorFormat color_format)`

| 功能   | 打印色彩格式描述                                                                                |
|------|-----------------------------------------------------------------------------------------|
| 输入参数 | <code>BmVpuColorFormat color_format</code> - 显示给定的色彩模式的描述,0 代表 I420;1 代表 NV21;2 代表 IF22 |
| 返回值  | 对应色彩的文字描述                                                                               |
| 函数说明 | 无                                                                                       |

13. `char const *bmvpu_frame_type_string(BmVpuFrameType frame_type)`

| 功能   | 打印帧类型描述                                                            |
|------|--------------------------------------------------------------------|
| 输入参数 | <code>BmVpuFrameType frame_type</code> - 帧类型的数据结构, 主要有 I、P、B、IDR 帧 |
| 返回值  | 对应帧类型的文字描述                                                         |
| 函数说明 | 无                                                                  |

14. `char const *bmvpu_enc_error_string(BmVpuEncReturnCodes code)`

| 功能   | 根据返回值, 打印错误代码描述                                 |
|------|-------------------------------------------------|
| 输入参数 | <code>BmVpuEncReturnCodes code</code> - 解码器的返回值 |
| 返回值  | <code>code</code> 对应错误的文字描述                     |
| 函数说明 | 无                                               |

15. `int bmvpu_enc_send_frame(BmVpuEncoder *encoder, BmVpuRawFrame const *raw_frame, bool isframe_end)`

|      |                                                                                                   |
|------|---------------------------------------------------------------------------------------------------|
| 功能   | 使用给定的编码参数对输入帧进行编码，并将编码后的输出帧信息存储在一个由用户提供的函数分配的缓冲区中                                                 |
| 输入参数 | BmVpuEncoder *encoder - 编码器<br>BmVpuRawFrame const *raw_frame - 输入帧<br>bool isframe_end - 最后一帧的标识 |
| 返回值  | BM_SUCCESS - 成功 else - 失败                                                                         |
| 函数说明 | 无                                                                                                 |

16. `int bmvpu_enc_get_stream(BmVpuEncoder *encoder, BmVpuEncodedFrame *encoded_frame)`

|      |                                                                                          |
|------|------------------------------------------------------------------------------------------|
| 功能   | 获得编码后的视频流并存入结构体 BmVpuEncodedFrame                                                        |
| 输入参数 | BmVpuEncoder *encoder - 编码器<br>BmVpuEncodedFrame *encoded_frame - 用于接收编码后的所有视频帧, 指针指向第一帧 |
| 返回值  | BM_SUCCESS - 成功 else - 失败                                                                |
| 函数说明 | 地址指针保存编码后第一帧图像的地址                                                                        |

17. `int bmvpu_enc_set_roiinfo(BmVpuEncoder *encoder)`

|      |                                 |
|------|---------------------------------|
| 功能   | 设置 ROI(Region of Interest) 相关信息 |
| 输入参数 | BmVpuEncoder *encoder - 编码器     |
| 返回值  | BM_SUCCESS - 成功 else - 失败       |

18. `int bmvpu_enc_ioctl_mmap(BmVpuEncoder *encoder, unsigned long addr, unsigned int size, unsigned long long *va)`

|      |                                                                                                                                      |
|------|--------------------------------------------------------------------------------------------------------------------------------------|
| 功能   | 地址映射接口                                                                                                                               |
| 输入参数 | BmVpuEncoder *encoder - 编码器<br>unsigned long addr - 设备内存的起始地址<br>unsigned int size - 映射内存大小<br>unsigned long long *va - 映射后的系统内存虚拟地址 |
| 返回值  | BM_SUCCESS - 成功 else - 失败                                                                                                            |
| 函数说明 | 为系统内存创建虚拟地址方便主机对内存块进行操作                                                                                                              |